

Abstract

Galeotti, John Michael. The EvBot: A Small Autonomous Mobile Robot for the Study of Evolutionary Algorithms in Distributed Robotics. (Under the direction of Edward Grant.)

This research has developed a new distributed robot-colony research platform that is robust and powerful. The colony is comprised of small individual autonomous mobile robots that are capable of providing a wealth of data and running complex controllers locally, autonomously, and remotely. The robots have been named EvBots for their focus on evolutionary robotics. Colony experiments are made possible in part by each EvBot's high-bandwidth communications link that allows communication and collaboration between colony members. Each EvBot is approximately eight inches in diameter and its hardware costs less than \$1000. An EvBot is capable of operating for over 2.5 hours using a 7.2V 3000 mAh battery. A custom Linux distribution was developed for the EvBot to enable it to run MATLAB, as well as other high-level software applications. The MATLAB high-level language allows new controllers to be quickly developed and easily simulated on desktop computers prior to being loaded onto physical EvBots. Experiments have shown that the EvBot can run MATLAB-based artificial-neural-network controllers with little or no modification to the controller following simulation. Experiments have also demonstrated the EvBot's plug-and-play vision and communications capabilities, all using MATLAB-based controllers.

THE EVBOT
A SMALL AUTONOMOUS MOBILE ROBOT FOR THE STUDY OF
EVOLUTIONARY ALGORITHMS IN DISTRIBUTED ROBOTICS

by
JOHN M. GALEOTTI


A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

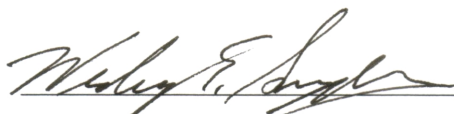


ELECTRICAL AND COMPUTER ENGINEERING

Raleigh

March 2002

APPROVED BY:


Chair of Advisory Committee

Biography

John Galeotti was born January 26, 1979 in Springfield, Missouri to Gary and Sharon Galeotti. He received his Bachelor of Science in Computer Engineering from North Carolina State University, Raleigh, NC in 2001, where he graduated as Valedictorian. He received his Master of Science degree in Computer Engineering from North Carolina State University, Raleigh, NC in 2002. He is a member of Phi Kappa Phi, IEEE, and the IEEE Computer Society.

Acknowledgments

I thank my Lord and Savior Jesus Christ for making the ultimate sacrifice, so that I might live. He gives meaning to my life and work, and He sustains me through difficult times. Without Him, this thesis would never have been written. I also thank my parents for their endless love and encouragement through the years. They have cultivated within me a curiosity and a love for knowledge for which I am grateful.

I would like to thank my advisor, Dr. Edward Grant, for his confidence in me and for the assistance he provided through my graduate research. His guidance is appreciated. I am also grateful to my committee, Dr. Wesley E. Snyder, Dr. Mark White, and Dr. H. Troy Nagle, for their patience and support.

I wish to thank all of the members of the Center for Robotics and Intelligent Machines for their friendship and support. I would especially like to thank Andrew Nelson, Greg Barlow, and Tony Frits for their help in conducting experiments.

Table of Contents

List of Figures	vii
List of Tables	xi
List of Abbreviations	xii
Chapter 1. Introduction	1
Section 1.1 Motivation.....	1
Section 1.2 Thesis Goals.....	2
Section 1.3 Outline of Thesis.....	3
Chapter 2. Literature Review	4
Chapter 3. Jumper Emulator (JE) Design	9
Section 3.1 Hardware Design	10
Section 3.1.1 <i>Treaded Base</i>	11
Section 3.1.1.1 Basic Stamp II (BS2) MCU.....	12
Section 3.1.1.2 H-Bridge Motor Drivers.....	12
Section 3.1.1.3 Wireless RS232 Receiver	14
Section 3.1.2 <i>Camera System</i>	15
Section 3.1.2.1 Very Small B/W CMOS Camera with Analog Output.....	16
Section 3.1.2.2 Analog Wireless Video Transmitter and Receiver.....	17
Section 3.2 Software Design.....	18
Section 3.2.1 <i>Wireless Communication Protocol</i>	18
Section 3.2.2 <i>Basic Stamp II (BS2) Code</i>	19
Section 3.2.2.1 Receive-Commands Mode	19
Section 3.2.2.2 Drive-Robot Mode.....	19
Section 3.2.3 <i>MATLAB Code</i>	20
Section 3.2.3.1 Image Acquisition	21
Section 3.2.3.2 Image Processing.....	21
Section 3.2.3.3 Image Understanding.....	23
Section 3.3 Initial Experiments and Results: Room Wandering	24
Section 3.3.1 <i>Obstacle Avoidance</i>	24
Section 3.3.2 <i>Shortcomings of the Visual Navigation System</i>	24
Section 3.3.2.1 Analog Transmitter Difficulties.....	25
Section 3.3.3 <i>Inherent Limitations of the JE</i>	26
Chapter 4. Evolutionary Robot (EvBot) Design	28
Section 4.1 Fundamental EvBot Requirements	28
Section 4.2 Derived EvBot Requirements	31
Section 4.3 Hardware Design	35
Section 4.3.1 <i>PC104 Stack</i>	37
Section 4.3.1.1 CPU, RAM, and I/O.....	38
Section 4.3.1.2 Flash Storage	39
Section 4.3.1.3 Wireless Ethernet.....	40

Section 4.3.1.4	USB Web Cam.....	40
Section 4.3.2	<i>Treaded Base</i>	41
Section 4.3.2.1	BasicX MCU.....	43
Section 4.3.2.2	Efficient H-Bridge PWM drivers.....	44
Section 4.3.2.3	Encoder Support.....	45
Section 4.3.3	<i>Utility Board</i>	46
Section 4.3.3.1	Power Supply and Battery.....	46
Section 4.3.3.2	Utility Devices and Interface.....	46
Section 4.3.4	<i>Support Hardware</i>	47
Section 4.4	Software Design	47
Section 4.4.1	<i>Treaded Base Interface Protocol</i>	48
Section 4.4.2	<i>Method for Adding a New Device</i>	49
Section 4.4.2.1	TB Firmware Modification.....	49
Section 4.4.2.2	Linux TB-Interface System Modification.....	50
Section 4.4.3	<i>Linux Distribution</i>	51
Section 4.4.3.1	Need for Linux.....	51
Section 4.4.3.2	Nature of Linux.....	51
Section 4.4.3.3	The Infinite Atom Linux Distribution.....	52
Section 4.4.3.3.1	<i>Full Wireless Ethernet</i>	54
Section 4.4.3.3.2	<i>Complete Imaging System</i>	55
Section 4.4.3.3.3	<i>Full MATLAB Support</i>	56
Chapter 5.	Experiments and Results	59
Section 5.1	The Maze Environment.....	59
Section 5.2	Neural Networks and Touch Sensors.....	61
Section 5.3	Visually Segregating Robots.....	64
Section 5.4	EvBots that Communicate.....	67
Section 5.5	Comparison to Similar Platforms.....	70
Section 5.5.1	<i>More Autonomous than the Rascal</i>	70
Section 5.5.2	<i>Smaller than the Tank</i>	71
Chapter 6.	Conclusion and Future Research	73
Section 6.1	Concluding Remarks.....	73
Section 6.2	Future Research.....	74
Chapter 7.	References	75
Chapter 8.	Appendices	80
Section 8.1	Jumper Emulator (JE).....	80
Section 8.1.1	<i>Hardware</i>	80
Section 8.1.2	<i>Software</i>	82
Section 8.1.2.1	JE Mobile Robot Source Code.....	82
Section 8.1.2.1.1	<i>Description and Explanation</i>	82
Section 8.1.2.1.2	<i>BS2 Source Code</i>	84
Section 8.1.2.2	Controlling-Computer Source Code.....	86
Section 8.1.2.2.1	<i>Description and Explanation</i>	86
Section 8.1.2.2.2	<i>MATLAB Source Code</i>	88
Section 8.1.2.2.3	<i>MATLAB-Support Source Code</i>	89
Section 8.2	EvBot Construction.....	90
Section 8.2.1	<i>Treaded Base (TB)</i>	90

Section 8.2.1.1	Construction	90
Section 8.2.1.2	Test Procedure	95
Section 8.2.2	<i>Utility Board (UB)</i>	96
Section 8.2.2.1	Construction	96
Section 8.2.2.2	Test Procedure	98
Section 8.2.2.3	Connection Cables	99
Section 8.2.2.4	Connection to the Treaded Base	100
Section 8.2.3	<i>PC/104 Stack</i>	102
Section 8.2.3.1	Component Preparation	102
Section 8.2.3.2	Stack Assembly	104
Section 8.3	EvBot Software	108
Section 8.3.1	<i>TB Source Code</i>	108
Section 8.3.1.1	Description and Explanation	108
Section 8.3.1.2	BasicX Code	109
Section 8.3.2	<i>Infinite Atom Linux Distribution</i>	120
Section 8.3.2.1	Description and Explanation	120
Section 8.3.2.1.1	<i>Basic EvBot Control Commands</i>	121
Section 8.3.2.1.2	<i>Ram Disk for Temporary Storage</i>	122
Section 8.3.2.1.3	<i>Linux Startup Sequence Customizations</i>	122
Section 8.3.2.1.4	<i>Modifications to LILO</i>	125
Section 8.3.2.1.5	<i>Modifications to the Kernel</i>	125
Section 8.3.2.1.6	<i>Redesign of the Initial RAM Disk</i>	126
Section 8.3.2.1.7	<i>Modification of the Init Scripts</i>	128
Section 8.3.2.1.8	<i>MATLAB Customizations</i>	129
Section 8.3.2.2	Linuxrc C Source Code	129
Section 8.3.2.3	/etc/inittab	130
Section 8.3.2.4	/etc/rc.d/rc.sysinit	131
Section 8.3.2.5	/usr/etc/rc.local	135
Section 8.3.2.6	Linux TB-Interface Utilities C Source Code	136
Section 8.3.2.6.1	<i>Basicx_interface</i>	136
Section 8.3.2.6.2	<i>Readpin</i>	137
Section 8.3.2.6.3	<i>Setpwm</i>	138
Section 8.3.2.7	Other Source Code and Binary Files	139
Section 8.3.3	<i>EvBot MATLAB Controller Source Code</i>	139
Section 8.4	EvBot Support Environment	140
Section 8.4.1	<i>EvBot Development Station</i>	140
Section 8.4.2	<i>Server Setup</i>	143
Section 8.4.3	<i>Maze Environment Detailed Description</i>	144
Section 8.4.4	<i>Simulation Environment</i>	146
Section 8.5	Setup and Configuration Procedure for a New EvBot	148
Section 8.5.1	<i>Creation of the DOC 5.0 Update Disk</i>	150
Section 8.5.2	<i>ESCD Reset Procedure</i>	151
Section 8.6	Procedure for Adding a New Controller to an EvBot	152

List of Figures

Figure 3.1 The JE (left) and its controlling computer with wireless interfaces stacked on top (right)	10
Figure 3.2 Flow of information diagram for the Jumper Emulator	10
Figure 3.3 The base of the tank-like toy vehicle (left) with populated proto board (right)	11
Figure 3.4 H-Bridge schematic.....	13
Figure 3.5 Wireless RS232 transmitter (top) and receiver (bottom) shown with a quarter for size comparison.....	14
Figure 3.6 Wireless RS232 transmitter (top right) installed on a computer interface board	15
Figure 3.7 JE camera system components that ride on top of the TB showing the camera (circled in red) and the transmitter (circled in bright green) mounted to their carrying harness.....	15
Figure 3.8 The JE's camera shown behind a quarter for size comparison	16
Figure 3.9 JE transmitter (left pane) and complete camera system (right pane, showing the TB-mounted components on the left and the controlling computer's video receiver on the right).....	17
Figure 3.10 JE with its metal box open, showing the proto board (and receiver) inside.....	18
Figure 3.11 JE controlling-computer flow-chart	20
Figure 3.12 Image captured by JE and processed in Figure 3.13.	21
Figure 3.13 MATLAB's analysis of the image presented in Figure 3.12 showing the user on a single screen the results of several operations applied to both the entire input image and just the bottom 2/5 of it, which corresponds to the floor area in front of the JE.....	22
Figure 4.1 Compact USB hub shown beside the EvBot's USB camera for size comparison.....	35

Figure 4.2 EvBot hardware diagram.....	36
Figure 4.3 EvBot equipped with camera and with treads removed.....	36
Figure 4.4 PC/104 modules connect electronically using a stacking version of the ISA bus, boxed in red on the left, and mechanically using 0.625 inch standoffs, shown on the right.....	37
Figure 4.5 The EvBot’s Treaded Base.....	42
Figure 4.6 Treaded Base close-up.....	43
Figure 4.7 CMOS inverter-based high-efficiency H-bridge.....	45
Figure 5.1 The EvBots’ maze with some of the visual barrier removed.....	60
Figure 5.2 EvBot equipped with touch-sensitive whiskers.....	62
Figure 5.3 Path taken by a genetically trained EvBot (middle right) as it navigated through a maze using only tactile sensors.....	62
Figure 5.4 An EvBot that bumped into a wall without triggering any of its tactile sensors.....	64
Figure 5.5 EvBot with red cylindrical shell for visual identification by other EvBots.....	65
Figure 5.6 This image of an EvBot was taken during an experiment by another EvBot that automatically saved this image to a server for later review and analysis by the human operators.....	66
Figure 5.7 An EvBot that cannot see that it is stuck.....	67
Figure 5.8 Two red EvBots (top left) clustered together and facing each other after taking the indicated paths through the maze in search of one another.....	68
Figure 5.9 EvBots that do not communicate must chase each other until an obstruction makes the lead robot turn to face the chasing robot, causing them both to cluster together.....	69
Figure 5.10 EvBots that do communicate cease to move away from each other as soon as one EvBot spots the other, causing the “lead” EvBot to gradually turn around until it finds the “chasing” EvBot.....	70
Figure 8.1 Component layout on the proto board.....	81

Figure 8.2 JE Schematic for the layout shown above in Figure 8.1 (The receiver attaches to P0.).....	81
Figure 8.3 JE mobile robot source code flowchart	82
Figure 8.4 JE controlling-computer flow-chart	86
Figure 8.5 MATLAB's analysis of the image presented in Figure 3.12 showing the user (all at once) the results of several operations applied to both the entire input image and just the bottom 2/5 of it, which corresponds to the floor in front of the JE.....	87
Figure 8.6 The original tread-drive base from a Rokenbok toy.....	90
Figure 8.7 The EvBot's Treaded Base as viewed from the side and from the top	91
Figure 8.8 Custom PCB with labeled components as seen from the top and bottom, with and without the BasicX socketed	92
Figure 8.9 Custom PCB layout (bottom copper viewed from top) with labeled power connections	92
Figure 8.10 Schematic for optionally sharing the BasicX's single IRQ between two encoders outputs.....	93
Figure 8.11 Treaded Base Plexiglas cover.....	94
Figure 8.12 Treaded Base power switch mounting bracket (all dimensions are in inches).....	94
Figure 8.13 Treaded Base power switch mounting and wiring connections	94
Figure 8.14 PWM waveforms on BasicX connection pins 5 through 8 of the TB (connected to channels 1 through 4 on the oscilloscope)	96
Figure 8.15 Utility Board PCB layout (bottom copper viewed from top) with labeled components.....	97
Figure 8.16 RS232 serial cable layout	99
Figure 8.17 Electrical power connections between the Utility Board and the Treaded Base.....	100
Figure 8.18 Utility Board support hole locations	101
Figure 8.19 Wire placement for connecting the Utility Board to the Treaded Base ..	102

Figure 8.20 MZ104 with labeled DOC and SDRAM.....	103
Figure 8.21 Camera support structures	104
Figure 8.22 EvBot prior to assembling PC/104 stack.....	105
Figure 8.23 EvBot with support components for the PC/104 stack.....	105
Figure 8.24 Port side of EvBot showing one view of wire connections to the MZ104 motherboard	106
Figure 8.25 EvBot with MZ104 motherboard mounted and wires connected.....	106
Figure 8.26 EvBot with camera mounting plate and PC-Card interface installed (before and after inserting the wireless Ethernet PC-Card).....	107
Figure 8.27 EvBot fully assembled with USB camera mounted	107
Figure 8.28 TB BasicX firmware high-level flowchart for the main loop	109
Figure 8.29 PC/104 stack high-level flowchart for the bootup sequence	124
Figure 8.30 The maze environment	145
Figure 8.31 Image of the maze acquired by its overhead camera during an experiment.....	146
Figure 8.32 The MATLAB simulation demo showing a single EvBot navigating through a maze	147

List of Tables

Table 3.1 H-Bridge Motor Control Outputs13

Table 3.2 Commands Supported by the JE Wireless Control Protocol19

Table 3.3 Sequence of BS2 Actions for Each Command Received20

Table 8.1 Commands Supported by the JE Wireless Control Protocol84

List of Abbreviations

- ADC = Analog to Digital Converter
- AI = Artificial Intelligence
- AM = Amplitude Modulation
- ANN = Artificial Neural Network
- ATA = Advanced Technology bus Attachment
- BIOS = Basic Input/Output System
- BS2 = Basic Stamp II
- CRIM = Center for Robotics and Intelligent Machines
- DARPA = Defense Advanced Research Projects Agency
- DHCP = Dynamic Host Configuration Protocol
- DIP = Dual Inline Package
- DOC = DiskOnChip
- DPDT = Double-Pole, Double-Throw
- EEPROM = Electronically Erasable Programmable Read-Only Memory
- ESCD = Extended System Configuration Data
- F/F = Female/Female
- FSM = Finite State Machine
- GPL = General Public License
- GUI = Graphical User Interface
- HTTP = HyperText Transfer Protocol

- I/O = Input/Output
- IEEE = Institute of Electrical and Electronics Engineers
- IRQ = Interrupt ReQuest
- JE = Jumper Emulator
- KVM = Keyboard, Video, Mouse
- LDO = Low Drop Out
- LILO = LInux LOader
- LOG = Laplacian Of Gaussian
- M/F = Male/Female
- MCU = Microprocessor Control Unit / microcontroller
- MSOP = Mini(Micro) Small Outline Package
- NCSU = North Carolina State University / NC State
- NTSC = National Television System Committee
- OHCI = Open Host Controller Interface
- PCB = Printed Circuit Board
- PCMCIA = Personal Computer Memory Card International Association
- PnP = Plug-and-Play
- PWM = Pulse Width Modulation
- RF = Radio Frequency
- RTC = Real Time Clock
- SDRAM = Synchronous Dynamic Random Access Memory

- SO-DIMM = Small Outline Dual Inline Memory Module
- SIP = Single Inline Package
- SPI = Serial Peripheral Interface
- SSH = Secure SHell
- TB = Treaded Base
- TBIP = Treaded Base Interface Protocol
- UART = Universal Asynchronous Receiver-Transmitter
- UAV = Unmanned Aerial Vehicle
- UB = Utility Board
- USB = Universal Serial Bus

Chapter 1. Introduction

Section 1.1 Motivation

Most robotic platforms available for research in the area of robot colony behaviors belong to one of two groups. The first group, those with powerful CPUs and advanced sensing capabilities, are both cumbersome and expensive (e.g. those produced by iRobot and Cybermotion). Those in the second group are affordable and easily maintainable (e.g. LEGO Mindstorms), but have such limited capabilities that they must often be remotely controlled via a central computer to implement evolved controllers of moderate complexity (i.e., an artificial neural network with a few hundred neurons). While remote control is feasible for small numbers of robots, a constrained communications bandwidth effectively limits the number of robots that can be remotely controlled in a single colony. Therefore, the research here has concentrated on developing a new robotic platform that is small and inexpensive and yet robust and powerful. The robot was designed to provide a wealth of data and to run complex controllers onboard. The robot for such a platform should be useful both by itself and within a colony setting; each robot's autonomous nature and onboard communications should enable complex robot interactions that are limited with current small robot platforms. To be truly useful, comprehensive development tools must be available for the robot colony, supporting both rapid controller development and robot simulation. This thesis presents the design and development of a new robotic platform capable of advanced experiments in colony behaviors and evolutionary robotics: the "EvBot"

platform. The initial research leading to the design and development of the EvBot is presented. In later chapters the EvBot's hardware and software design will be explained in depth and demonstrations of the EvBot's current potential will be demonstrated. Lastly, thought will be given to how the EvBot platform can be expanded and enhanced in the future.

Section 1.2 Thesis Goals

The objectives of this thesis are to describe the:

- Design and construction of the EvBot's predecessor: the Jumper Emulator (JE), a simple robot to emulate a jumping robot's surveillance capabilities.
- Design and construction of an EvBot: a small, robust robot capable of acting autonomously, providing a wealth of data, and engaging in complex interactions with other robots in a colony environment.
- Development and implementation of EvBot software, including the production of a very small Linux distribution that supports EvBot control and communication.
- Demonstration of the EvBot's ease of use and its advanced capabilities which allow it to run onboard MATLAB-based controllers and perform complex robotic colony experiments.

Section 1.3 Outline of Thesis

In Chapter 2, the literature is reviewed. An extensive summary of robotic platforms that are in active use in the areas of colony behaviors and evolutionary robotics is presented.

In Chapter 3, a simple robot is described at the hardware and software levels. Experiments carried out using this prototype robot highlighted the need for a new robotic platform design.

In Chapter 4, the EvBot is presented. The requirements imposed upon its design are put forward and analyzed, leading to an optimized hardware and software design.

In Chapter 5, an experimental test bed consisting of a maze environment is described. Three main experiments are performed using the EvBot platform, and the results are presented.

In Chapter 6, conclusions are drawn and the potential future research is discussed.

Finally, in the Appendices detailed descriptions of the EvBot's construction and software packages are presented.

Chapter 2. Literature Review

Autonomous robots have been an active research area ever since the neurophysiologist W. Grey Walter first used vacuum tubes as neurons in his robot “tortoises” [1]. Walter used his robots to show how seemingly complex behaviors, including group behaviors, could be produced by equipping each robot with a brain of only a few neurons. His discovery laid the foundation for evolvable robot controllers based on artificial neural networks. In the late 1980’s, when the artificial intelligence (AI) community at large stagnated in its attempts to advance AI in a disembodied form, Rodney Brooks (currently the director of MIT’s AI Laboratory) argued for the physical grounding hypothesis on which he based his subsumption architecture. This hypothesis states that an intelligent system must be grounded in the real world, because “the world is its own best model” [2]. During this period at the onset of interest in evolved robot controllers, all experiments were performed only in simulation. Brooks once again argued for the necessity of physical robot experiments and the inadequacy of simulation alone [3]. Nevertheless, at the end of 1998 Meyer counted only 30 published evolutionary robotics projects involving real robots. Many of these were based on the small and relatively simple Khepera robot [4]. Complex evolved controllers, such as those described in [5] and [6], existed only in simulations. Perhaps this was due in part to the inadequacy of robotic platforms. Robots with powerful CPUs and sensing capabilities are cumbersome and expensive. On the other hand, most smaller, affordable robots must be remotely controlled to implement any controller of moderate complexity.

One of the smallest and simplest robots used in the literature is the Alice [7], a wheeled robot controlled by a minimal PIC microcontroller. The very popular Khepera has been used for numerous experiments, both as an individual [8][9][10][11] and as part of a colony [12]. It was often pre-trained in simulation [13][14]. Part of the Khepera's popular appeal is apparently due to the excellent simulation environments that are available for it. The Khepera is a little larger than the Alice (55 mm diameter), and with its Motorola 68k series CPU, it is also a little more powerful. Still, with its limited computational ability, its limited RAM (256 Kbytes), and its need for an external camera to be accurately located (self-localized), most researchers elect to control (and power) it remotely from a host computer using a special aerial umbilical connected to the Khepera and suspended over its environment. Such a setup effectively restricts the number of Kheperas that can be used concurrently without eventually entangling their umbilicals. Such constraints may help explain why the DEMO research group at Brandeis University designed and developed their own colony of even larger robots that were powered from their floor for their online learning experiments [15]. The capabilities of these robots, however, were only on par with those of the Alice robot. Other simple wheeled robots larger than a Khepera include one built out of LEGOs [16] that also utilizes a Motorola 68k CPU and another built for colony experiments [17] that uses Motorola's simple 6811 CPU.

Walking robots have also been built for evolutionary robotics experiments. These range from the minimal Stiquito hexapod [18] to the larger (but still simple) Hermes II hexapod [19] and Jakobi's octopod [20]. Some of the larger walkers were used to run

artificial neural network (ANN) controllers containing on the order of 50 neurons. Still, controllers of more advanced complexity remain out of these robots' reach.

The minimal CPUs on all of these robots have encouraged some groups to abandon the use of onboard control altogether when designing their own custom robots. For example, the **Scout [21]** and **Clodbuster [22]** always rely on a host computer (which could be on a larger, master robot) to completely control all but their most simple hard real-time tasks, such as maintaining speed control.

Those researchers seeking truly autonomous robots capable of running moderately complex controllers, or the ability to simultaneously operate many robots equipped with high-bandwidth sensors, such as cameras, have resorted to using larger, more expensive robots. One exception is the **EyeBot family [23]** (described in [24]). EyeBots combine small cameras with moderately advanced custom hardware and a custom operating system. These non-standard custom elements can potentially restrict and complicate controller development due to a lack of standardized third-party development tools. Other groups have used Sony's AIBO walking robot [25] and have developed a simulation environment for it [26]. The AIBO includes an articulated color camera and a 32-bit RISC CPU, but it is as large as a small dog and requires a lot of power to operate. Furthermore, it is very expensive, making it hard to justify its use for colony experiments unless its life-like walking ability is required.

One of the smallest robots in the literature with an entire **onboard PC-compatible computer is the Rascal**, which is roughly one foot in diameter. PC-compatibility has the advantage of accelerating software development by providing a wealth of tools, readily

available software libraries, and a standards-compliant system. Winfield and Holland added wireless Ethernet to the Rascal [27], producing a platform suitable for modern research into robotic interaction. Their platform is not without fault, however. With “only” a 486 CPU and 4 MB of RAM, their platform was designed to be remotely controlled the majority of the time. Nevertheless, it is superior to most other remote controlled robots because many of the problems normally associated with remote control are alleviated by the high bandwidth and standards compatibility of wireless Ethernet. They consume much of their Rascal’s 80 MB IDE flash disk with their Linux distribution, however, leaving insufficient room for an advanced controller environment for the support of highly complex controllers. They do leave sufficient room, however, for their simple, onboard controller that merely receives and executes basic commands from a remote computer.

The Tank robots at Carnegie Mellon are almost twice the size of the Rascals [28]. The Tanks are often used to control a small fleet of much simpler robots known as the Millibots, which are a little larger and more powerful than the Khepera. Like the Khepera, the Millibots are not capable of running truly autonomous complex controllers, and are most frequently used as distributed sensor platforms for a Tank, which must act as their team leader. Although the Tanks are meant to act autonomously, they still suffer from many of the same deficiencies as the Rascal, despite their increased size. They have a 486 CPU and a large software package that resides on a real hard drive. While a hard drive has sufficient storage space for very complex controllers, it is susceptible to shock damage and requires a significant amount of continuous power to operate.

There are many robots larger than the Tank that can be equipped with workstation class computers. Some walk [29] while others are equipped with wheels [30], and small numbers of such robots have been used for group experiments [31][32]. Unfortunately, these platforms are too large and expensive to be practical for many research applications. Hence, the literature has shown that there is a need for a small and inexpensive robotic colony research platform that is computationally powerful, capable of autonomous onboard or local control, and able to provide a wealth of data.

Chapter 3. Jumper Emulator (JE) Design

The current EvBot design was preceded by a much simpler robot—the **Jumper Emulator (JE)**—with capabilities similar to (but simpler than) the Khepera [11] or Scout [21]. The JE was not intended to actually jump. Rather, it was developed to show, via emulation, the expected intelligence-gathering capabilities of a small jumping robot that was being developed in the Center for Robotics and Intelligent Machines (CRIM) at North Carolina State University (NC State). The small jumping robot was to be actuated by novel solid-state, high-displacement, high torque, piezoelectric transducer-based actuators that the CRIM was developing as part of a DARPA-sponsored research program (#N3998-98-C3536). Nevertheless, even with such actuators, the jumping robot would have stringent size, weight, and power constraints on the equipment it could carry, potentially restricting its capabilities. Therefore, DARPA required that the JE be built to demonstrate the expected reconnaissance capabilities of the jumping robot. As such, all sensors and control hardware on the JE also had to be small, lightweight, and draw little power. Alternatively, the JE could be equipped with sensors and hardware not meeting the size, weight, and power requirements if the availability of functionally identical equipment meeting those requirements was expected in the near future. The JE had to be capable of image acquisition and image interpretation for autonomous robot navigation and control. It was also desirable to be able to use sound, thermal, and chemical sensors. The only principal feature/limitation of the jumping robot that was not represented in the

JE was jumping locomotion capability; the JE could only drive around using a small treaded base (TB).

Section 3.1 Hardware Design

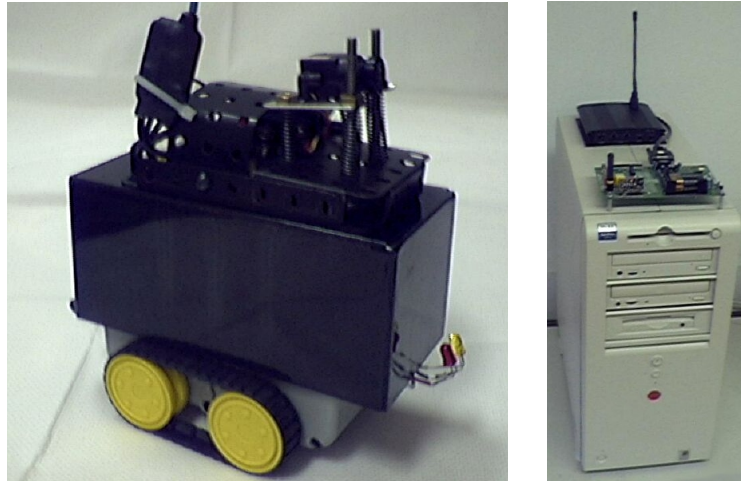


Figure 3.1 The JE (left) and its controlling computer with wireless interfaces stacked on top (right)

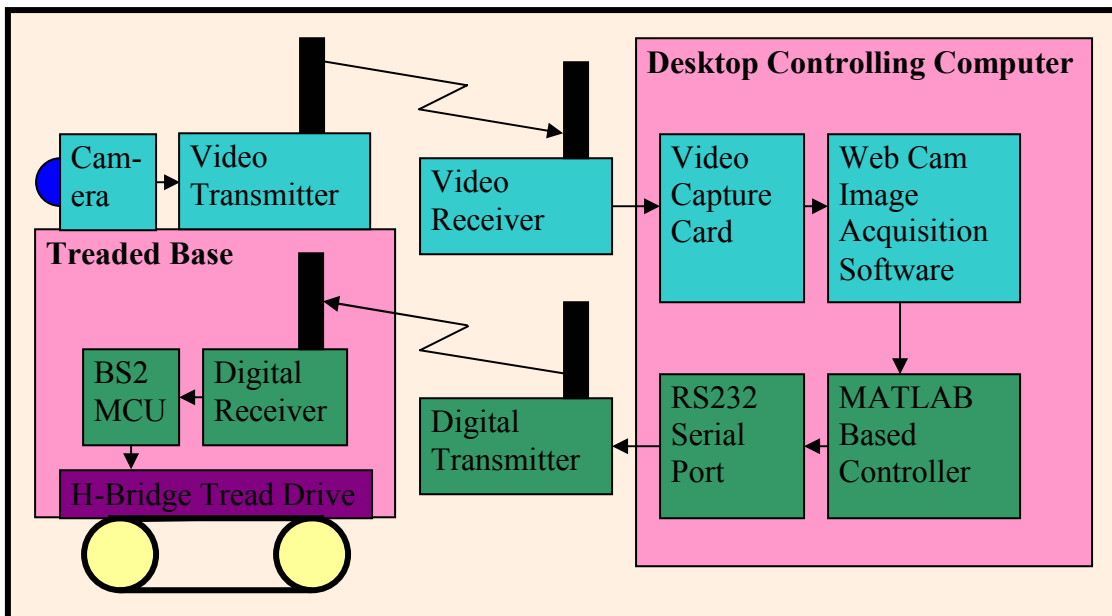


Figure 3.2 Flow of information diagram for the Jumper Emulator

To accelerate development of the JE, it was designed to be remotely controlled by a desktop computer (although the control algorithms were kept simple enough to operate within the expected capabilities of the final jumping robot's onboard hardware). Two separate wireless links were used to implement the remote control capability, one each for image and command transmission (allowing the onboard camera system to be completely detached from the TB and its associated robot body).

Section 3.1.1 Treaded Base

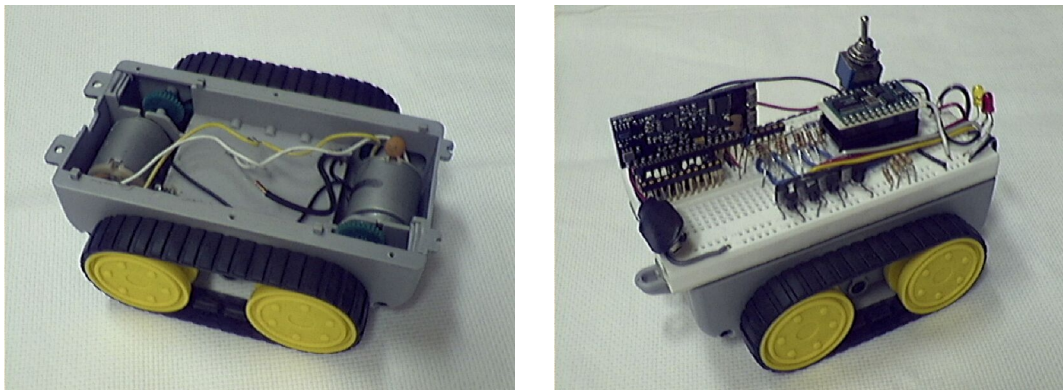


Figure 3.3 The base of the tank-like toy vehicle (left) with populated proto board (right)

The TB was built around the core of a small, self propelled tank-like toy vehicle. The shell and control hardware were removed, leaving only the base with its treads, DC motors, internal gearing, and battery compartment (Figure 3.3, left pane). A proto board was placed on top of the TB to hold the JE's electronic components (Figure 3.3, right pane). Those components can be divided into three primary categories: the Basic Stamp II (BS2) microcontroller (MCU), the H-bridge motor driving circuits, and the wireless RS232 receiver.

Section 3.1.1.1 Basic Stamp II (BS2) MCU

A Parallax BS2 was chosen for the TB's onboard microcontroller. It was responsible for processing all commands received over the wireless link and interpreting them into appropriate control signals for the H-bridge motor drivers. The BS2 is programmed in PBASIC2, which is Parallax's enhanced version of the BASIC programming language. PBASIC2 allows rapid control software development. The BS2 has 16 general purpose pins that can be independently programmed to function either as outputs or as inputs (either high-Z or pulled high). Four such pins were used to control the two motors' H-bridges. Unfortunately, the BS2 proved to be slow, and its 26 bytes of usable RAM proved to be insufficient. Both the BS2 and Parallax's more advanced BS2 SX suffer from an inability to multitask, making them incapable of processing interrupt requests which are necessary for many control actions that might otherwise have been implemented on the JE. In short, the BS2 must "stop" execution when waiting on serial data from the wireless receiver. This is because it does not have a hardware buffer to queue data received from a serial interface prior to processing. Nevertheless, it proved to be adequate for the purpose of executing the simple commands received over the wireless RS232 link and outputting appropriate logic values to the two H-bridges' inputs. More details about the BS2, and the JE's hardware in general, can be found in Section 8.1.1 in the Appendix.

Section 3.1.1.2 H-Bridge Motor Drivers

The H-bridge motor drivers (see Figure 3.4 for their schematic) directly powered the small DC motors that drove the treads. The primary purpose of each H-bridge was to

selectively supply the high current (approximately 100 mA) that was required by each motor while simultaneously drawing as little power as possible from its inputs, and thus from the MCU to which they were connected. Each H-bridge, one per motor, was capable of attaching each of its two outputs to either the motor-supply voltage or to ground. When a motor was attached across its two leads, an H-bridge could keep the motor idle or spin it in either direction, depending on the logic state of the H-bridge's input pins, as indicated in Table 3.1.

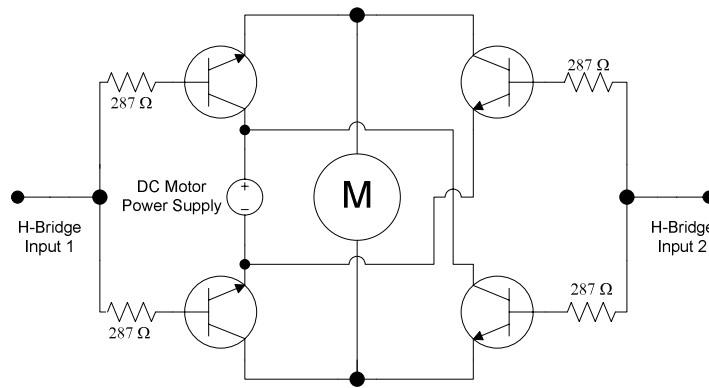


Figure 3.4 H-Bridge schematic

Table 3.1 H-Bridge Motor Control Outputs

H-Bridge Input 1	H-Bridge Input 2	Motor Rotation
high	high	idle
high	low	clockwise
low	high	counterclockwise
low	low	idle

The particular H-bridge circuit used on the JE was based on NPN transistors. As such, it required current limiting resistors on its transistors' inputs, and it suffered from a total drop of a few tenths of a volt across the H-bridge's transistors.

Section 3.1.1.3 Wireless RS232 Receiver

The small wireless RS232 receiver (Figure 3.5) on the TB was used to receive the commands from a controlling desktop computer that has an attached wireless RS232 transmitter (Figure 3.5 and Figure 3.6). Since RS232 is the standard protocol used by a desktop computer's serial ports, the wireless receiver allowed the MCU to function as if it was hard-wired to the desktop-computer's serial port. Additional software (Section 3.2.1) was required for minimal error checking, however, since RF interference and maximum transmission distances could corrupt transmitted data.

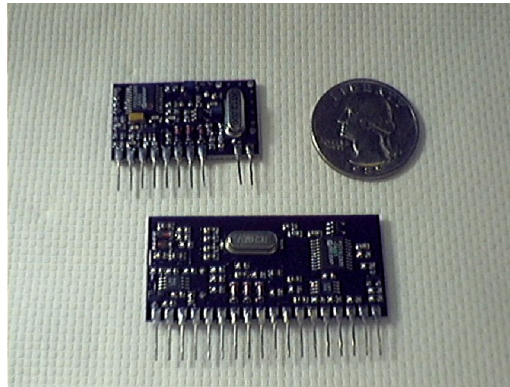


Figure 3.5 Wireless RS232 transmitter (top) and receiver (bottom) shown with a quarter for size comparison

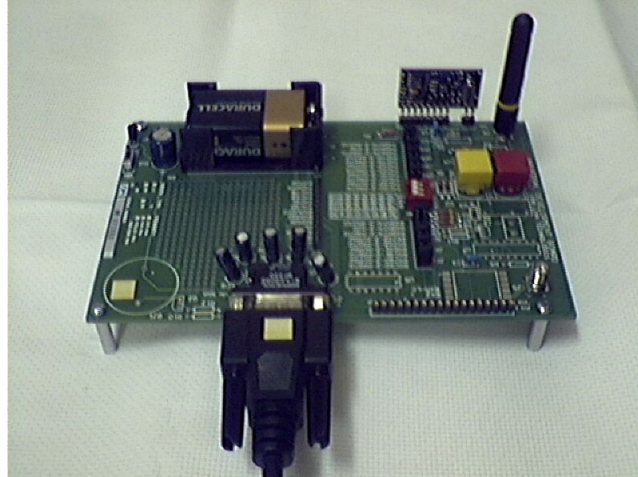


Figure 3.6 Wireless RS232 transmitter (top right) installed on a computer interface board

Linx Technologies produced the HP Series-II transmitter and receiver used by the JE. The HP Series-II was chosen for its long range of up to 1/4 mile open-air when used with its 1/4 wave whip antenna, its comparatively high bandwidth that supports serial rates up to 50 Kbps, and its user-selectable operating frequency. On the JE, the receiver's only antenna was its antenna-connection pin (in Figure 3.5, this is its right-most pin along the bottom). This pin alone proved adequate for reception from another room.

Section 3.1.2 Camera System

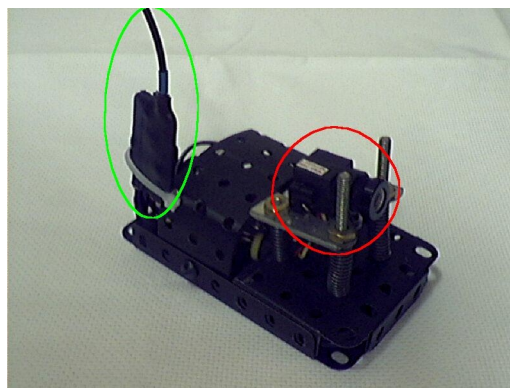


Figure 3.7 JE camera system components that ride on top of the TB showing the camera (circled in red) and the transmitter (circled in bright green) mounted to their carrying harness

The camera system on the JE (Figure 3.7) was a completely separate component from the TB on which it rode. It did not need to be connected to the base since it had its own power source and its images were processed remotely. The camera needed only to be positioned on top of the TB at the front to allow the JE to clearly see where it was headed. The camera was installed aimed slightly above the horizon. This positioning allowed the JE to see enough of the floor in front of it to navigate, while allowing a human operator, if present, to see as much of the JE's environment as possible. Human remote observation capability was important because the JE was designed to emulate a reconnaissance robot. The camera system continuously transmitted its analog output over a wireless link to a video capture card installed in the controlling computer.

Section 3.1.2.1 Very Small B/W CMOS Camera with Analog Output

The PC51XS Inline Micro Video Camera (produced by Supercircuits, shown in Figure 3.7 and Figure 3.8) was chosen for the JE due to its record-setting small size, as well as its low power requirements and acceptable field of view. It was only 0.6" square and 1.3" long, and yet it had 240 lines of grayscale resolution. The camera's output was a standard analog composite NTSC video signal.



Figure 3.8 The JE's camera shown behind a quarter for size comparison

Section 3.1.2.2 Analog Wireless Video Transmitter and Receiver

Supercircuits' AVX900-MINI-A was chosen for the wireless video transmitter due to its small size, and the matching AVX900RS was chosen as the receiver (Figure 3.9). The composite video signal was transmitted as an AM signal in the 900-930 MHz range. While the transmitter was quite small, as required for a miniature jumping robot, the receiver was too large to be carried onboard even a moderately small robot, were such desired. Also, as described in the experiments section, the out-of-band noise from the transmitter was so high that the TB had to carry its own electronics in a shielded box (Figure 3.10) to prevent the digital receiver from picking up enough noise on its attached wires to interfere with its operation. The AVX900RS receiver output the composite analog video signal received from the robot's camera into a standard video-capture-card in the controlling computer.

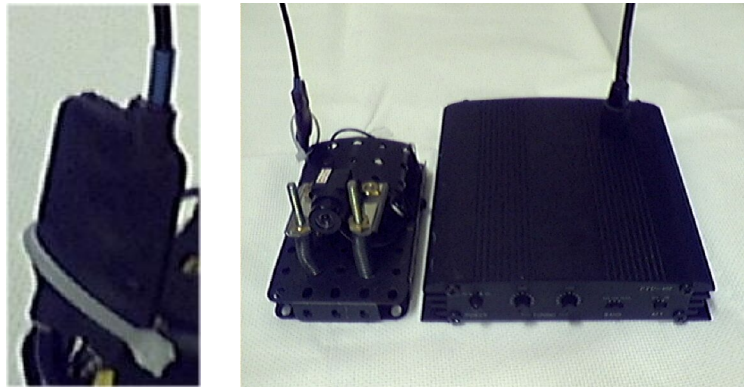


Figure 3.9 JE transmitter (left pane) and complete camera system (right pane, showing the TB-mounted components on the left and the controlling computer's video receiver on the right)

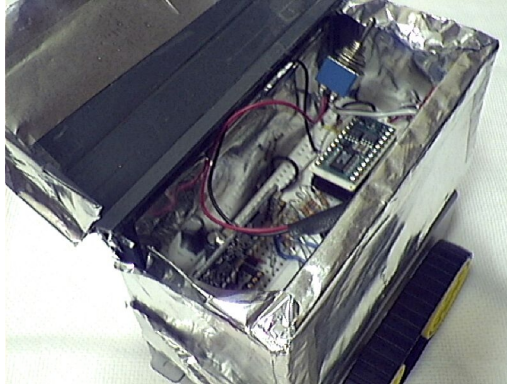


Figure 3.10 JE with its metal box open, showing the proto board (and receiver) inside

Section 3.2 Software Design

Section 3.2.1 Wireless Communication Protocol

The BS2's software is built around a digital wireless communication protocol used by the desktop computer to control the robot. The protocol was very simple, and although it only supported four commands, it could have easily been expanded to cover many more. There was only one software layer in the protocol and only one type of packet. Each transmitted packet began with the fixed header `0xFFFFFFFF80`, immediately followed by a single command byte, which terminated the packet. The first two bytes of the header were used to resynchronize the RS232 receiver's bit-slicer, which was used to pull digital data from a transmitted signal. The last two bytes marked the start of the packet data and were chosen to minimize the possibility of their random occurrence, as determined by the engineers at Linx Technologies. Four commands were implemented in the protocol. The command byte could be one of the values shown in Table 3.2.

Table 3.2 Commands Supported by the JE Wireless Control Protocol

Command Byte (Character Representation)	Command Description
F	Move forward for 1.5 seconds
B	Move backward for 1.5 seconds
L	Spin left for 0.4 seconds
R	Spin right for 0.4 seconds

Section 3.2.2 Basic Stamp II (BS2) Code

The BS2 code was responsible for reading commands received on its serial port (from the RS232 receiver) and actuating the TB's motors accordingly. It continuously looped between those two modes of operation. The entire source code for the Stamp is described and presented in Section 8.1.2.1.2 in the Appendix.

Section 3.2.2.1 Receive-Commands Mode

The BS2 continually scanned its RS232 serial port input for the sequence 0xFF80, followed by a single data byte. That data byte was assumed to be a command byte and was compared to all known command byte values. If there was a match, the program would jump to the command's corresponding PBASIC instructions. Otherwise, a warning LED on the JE was flashed before the BS2 resumed scanning its serial input.

Section 3.2.2.2 Drive-Robot Mode

The PBASIC code for each of the commands was very similar. For each command, the sequence shown in Table 3.3 was executed. Control was then returned to the BS2's main loop which resumed scanning the RS232 serial input.

Table 3.3 Sequence of BS2 Actions for Each Command Received

Set all the H-bridge motor-control pins such that the motors spin in the desired directions
Wait a fixed duration of time
Reset all the H-bridge motor-control pins to zero, stopping both of the motors.

Section 3.2.3 MATLAB Code

MATLAB, which ran on a desktop computer, completely controlled the JE's behavior using a knowledge-based controller. Figure 3.11 shows a flowchart for the controlling computer's MATLAB controller. Complete MATLAB code is described and presented in Section 8.1.2.2 in the Appendix.

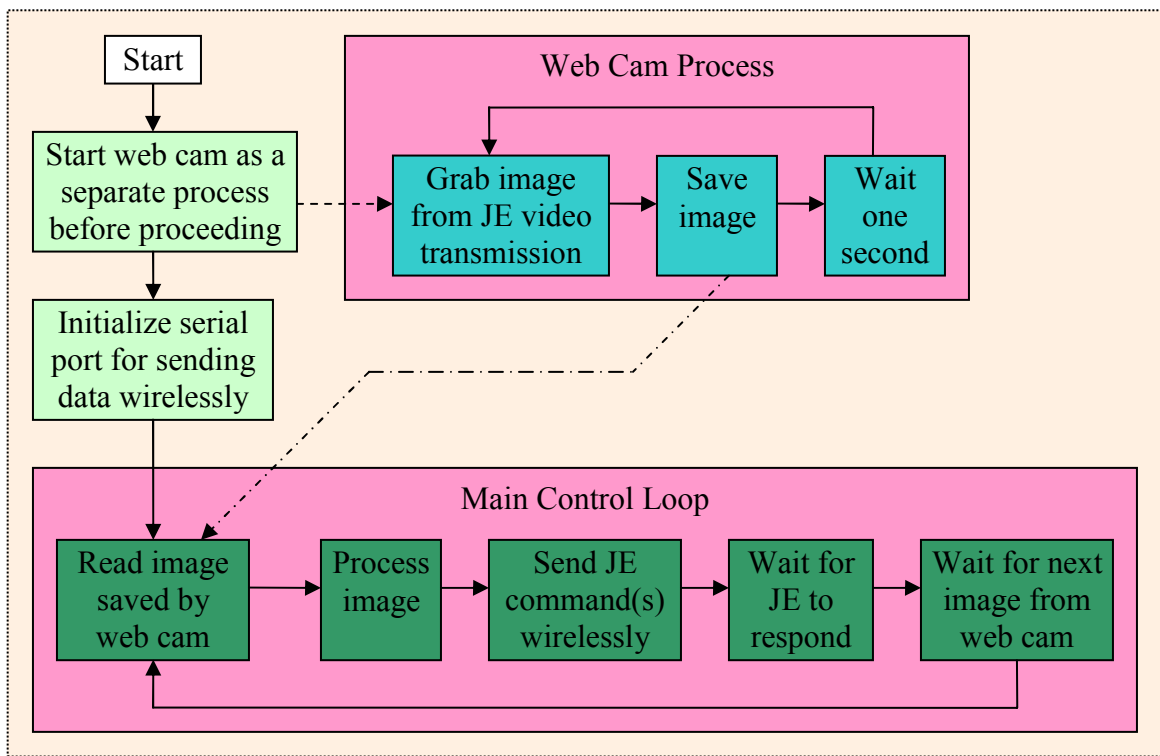


Figure 3.11 JE controlling-computer flow-chart

Section 3.2.3.1 Image Acquisition

MATLAB would begin by starting a standard web cam application. The web cam would run at the same time as MATLAB, grabbing the current image from the computer's video capture card once a second and saving it to a fixed filename. After initializing the computer's serial port, MATLAB would enter its main control loop where it would read the last image saved by the web cam and proceed to process it. Figure 3.12 is an example image the web cam acquired from the JE for MATLAB to process.



Figure 3.12 Image captured by JE and processed in Figure 3.13.

Section 3.2.3.2 Image Processing

MATLAB's image processing toolbox was used extensively to process the images. Several image measurements and interpretations would be shown on the screen to help the user adjust the controller (as shown in Figure 3.13), but edge detection on the bottom 2/5 of the image was all that was used to control the robot. Edge detection is an algorithm that looks for neighboring pixels in an image with strong differences in intensity, indicative of an actual edge between two different surfaces depicted in the

image. Typically, edge detection also uses some algorithm(s) both to prevent noise from introducing small false edges and also to prevent blur from masking larger, real edges. The specific edge detector used to control the JE was the Laplacian of Gaussian (LOG) edge detector included in MATLAB's image processing toolbox, although it was assumed that even simpler edge detectors could be made to work. In Figure 3.13, the center image second from the bottom shows the result of applying the LOG edge detector to the input image depicted in Figure 3.12. The center image on the bottom row shows the result of applying the LOG edge detector to only the bottom 2/5 of the input image.

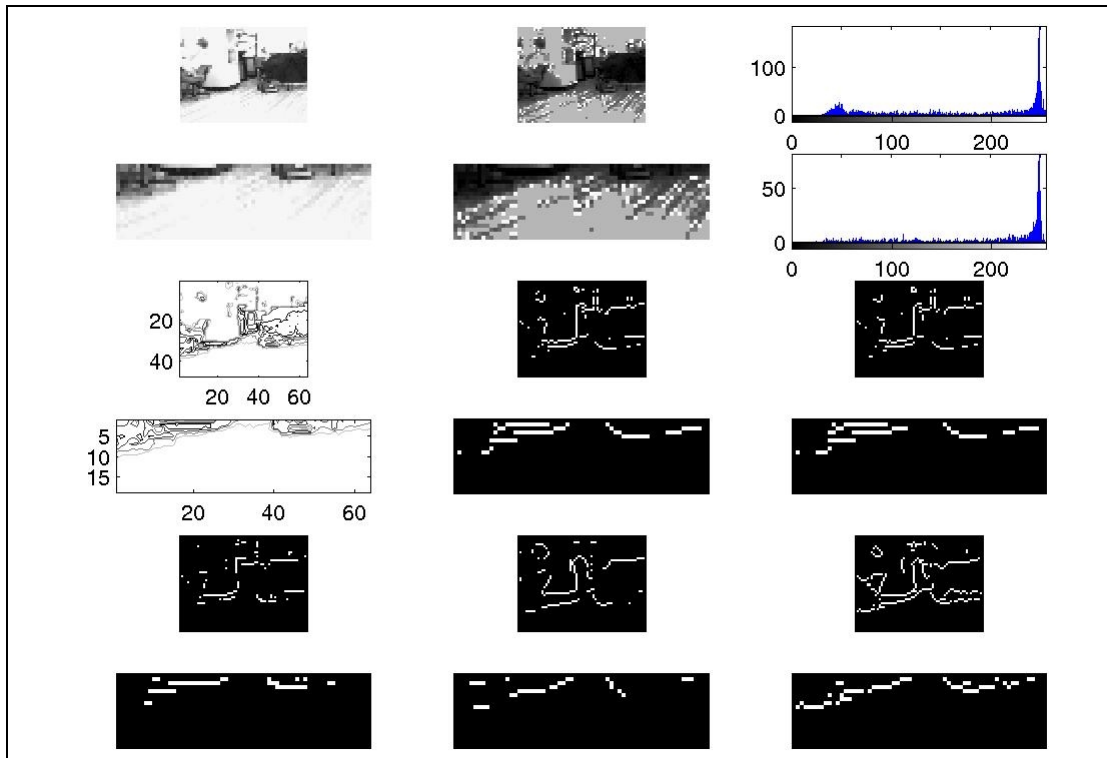


Figure 3.13 MATLAB's analysis of the image presented in Figure 3.12 showing the user on a single screen the results of several operations applied to both the entire input image and just the bottom 2/5 of it, which corresponds to the floor area in front of the JE

Section 3.2.3.3 Image Understanding

Once the edges had been found in the bottom 2/5 of the image, that portion of the image was even further partitioned into three adjacent regions corresponding to the floor areas immediately in front of the JE and to its left and right. These regions would normally contain any obstacles that may interfere with the JE's next movement. Such obstacles could usually be identified by the edges between themselves and the floor or walls around them. By counting the number of pixels marked as edges in each region, MATLAB could determine whether or not each region contained an obstacle and therefore decide if the JE should drive straight forward, turn and drive forward, or if it must back up before turning to face another (hopefully clear) direction. This simple process could make the JE wander around a room without getting itself trapped, while theoretically also being able to simultaneously use other algorithms to scan for surveillance targets, using vision and/or other sensors.

DOS batch file scripts were used to interface MATLAB with the computer's serial port. The scripts took a single character command as their argument, and then prefixed it with a packet header before transmitting it over the serial port. The wireless RS232 transmitter was connected to the serial port and relayed the entire packet to the JE's RS232 receiver and thus ultimately to its BS2 MCU.

Section 3.3 Initial Experiments and Results: Room Wandering

Section 3.3.1 Obstacle Avoidance

The JE was allowed to wander through various rooms of the CRIM. When the wireless links functioned properly, the JE would almost always successfully avoid obstacles and eventually cover the area of the room in which it was placed. The JE would also drive through doors (in the natural course of its “random” wandering) and explore other rooms. It would usually avoid driving under tables, due to the stark contrast between their shadows and the floor. Since the jumping robot in development was expected to jump up to a meter in height; such behavior would clearly benefit the robot by helping prevent “head injury.” With the addition of a short-range secondary obstacle sensor, the JE could probably have been made to hide under tables by reversing its control rule to move toward edges (such as those caused by shadows) with short jumps when the secondary sensor reported no obstacles where the edges indicated an obstacle existed.

Section 3.3.2 Shortcomings of the Visual Navigation System

Unfortunately, being equipped with only an inexpensive grayscale camera, the JE was unable to see brown cardboard boxes when placed on a hardwood floor. To the camera, the colors were almost identical. Also, because the video transmitter’s antenna was higher than the camera, the JE would occasionally get stuck with its antenna against a chair rung positioned just high enough for the rest of the JE’s body to pass under. The first problem could easily be solved by the addition of a short-range secondary obstacle

sensor. The second would have to be ignored in designing an actual jumping robot, which would be less likely to get stuck under a chair rung and would have to possess a mechanism for righting itself should it fall anyway. While not a problem for the JE, it should be pointed out that there was a considerable delay (one to two seconds) after the JE finished one command before MATLAB would send another. This delay was due primarily to image acquisition. The web cam software could capture one image a second at maximum speed, forcing MATLAB to wait over one second after the JE stopped before reading the web cam's most recently acquired image from disk. The lack of buffers in the JE (due to the JE's minimal RAM supply) also precluded many speed optimization methods, and the use of remote processing added a communications delay as well.

Section 3.3.2.1 Analog Transmitter Difficulties

As already alluded to, the wireless links would not always function correctly. The video receiver had to be hand tuned to the exact frequency of the video transmitter, and that frequency would drift, apparently as the transmitter's battery voltage decayed. Without readjusting the frequency of the receiver, the quality of the computer's captured images would slowly and continually decline, reaching an unacceptable level within just a few hours.

With their center frequencies located near each other, the wireless links were prone to interference despite documentation to the contrary. In particular, the analog video transmitter was believed to be generating large amounts of out-of-band noise. The digital link would function properly until the transmitter was turned on and placed near

the digital receiver, i.e. on the JE. The problem seemed to get worse as the video transmitter's battery voltage would drop (even within normal operating range), although there was a fair amount of case-by-case variance. Temporarily adding a regular antenna to the digital receiver did not help the problem. Unfortunately, the video transmitter and digital receivers had to be near each other out of the necessity for a small robot. As described in the Section 3.1.2.2, the problem was eventually overcome by surrounding all the circuitry (including the digital receiver) on the TB with a shielded metal box. The box usually provided adequate attenuation to block the video transmitter's out-of-band noise, while still passing an adequate level of the digital transmitter's signal for the JE to operate in a different room than the transmitter. However, when the video transmitter's battery voltage would drop below a certain point, the out-of-band noise would increase to a level strong enough to penetrate the shielded box, and once again the JE would have difficulty receiving commands from the controlling computer.

Section 3.3.3 Inherent Limitations of the JE

While the JE mostly fulfilled its own design requirements, meeting those stringent requirements imposed significant limitations on the JE for use with other research. In particular, the JE was not designed to work autonomously nor within a colony of robots. Its onboard processing power was very limited, its battery life was short, and it lacked the RAM to hold a single digital image. Multiple analog video transmitters like the one used on the JE cannot be used simultaneously in the same room. The JE's wireless digital throughput was too low for many types of inter-colony knowledge sharing. The JE was not even equipped for inter-robot communication. Nor could its existing digital

communications hardware have been readily expanded to handle multiple robots because of its limited channel spacing options, high power output, and lack of medium access control. The interface capabilities of the JE were also very limited, discouraging significant platform enhancement. These limitations highlighted the need for the new EvBot platform, whose design greatly benefited from insights gained while designing, building, and using the JE.

Chapter 4. Evolutionary Robot (EvBot) Design

Section 4.1 Fundamental EvBot Requirements

The EvBot design process was driven by five primary goals for the final EvBot robotic research platform. These goals were chosen to make the EvBot a useful research platform for a wide range of advanced experiments involving distributed mobile robots. In order to be useful by itself as well as in a colony, and for pure as well as applied research, the EvBot should have the following attributes:

- Be fully autonomous but capable of group behavior
- Be small and inexpensive
- Be robust
- Provide a wealth of data
- Provide a user-friendly development platform

Autonomy is a practical necessity for experiments with large numbers of robots and for robot use in remote locations such as deep space. Maintaining, controlling, and transporting a desktop or laptop computer for each EvBot (or group of EvBots) would be impractical for a large number of EvBots. Communications bandwidth is also quickly consumed if every EvBot in a colony must continually send all its data to a remote computer for processing. This is especially problematic for image and sound data, given their typically large bandwidth requirements. Therefore, each EvBot must be capable of running a local onboard controller that produces complex behaviors based on large amounts of input data. Autonomy in no way precludes colony behaviors, rather it aids in

their implementation. It can enable virtually unlimited simultaneous, complex local interactions between robots, provided the robots have short range communications ability; as long as two groups of EvBots are separated beyond their communication radius, each group can have its own “private” conversation without using any of the other group’s bandwidth. For example, most insect colonies are made of autonomous individuals that benefit from local interactions with other colony members, ultimately achieving division of labor and information sharing. It is hoped the EvBot will eventually be used to mimic the insect world in this regard. Yet, to be useful for the widest range of experiments, the EvBot should also be able to surrender partial and/or complete control to a remote computer (or computer cluster). This is a requirement in situations where enormous processing power is needed.

Small size is an absolute requirement in order to experiment with large numbers of robots in a small laboratory space. Therefore, the EvBot dimensions should be bounded by a relatively small circle, preferably of diameter less than nine inches. Smaller robots are typically less expensive than larger robots with equivalent computational and sensing abilities. Although making the EvBot larger (within a factor of 2) would probably yield a platform with a little more onboard-capability per dollar spent, the smaller EvBot should still be much less expensive than the several foot tall robots used in many experiments [30][32]. Because it would take many EvBots to build a single research colony, and most laboratories have limited funds, the components for an individual EvBot should cost as little as possible, preferably not more than \$1000 each.

Robustness in the face of unknown future requirements is crucial to the longevity of a robot-research platform. The EvBot should be as modular as possible. It should be possible to add a wide variety of new sensors and actuators, and it should be easy to remove old ones as well. New sensors may be simple passive components such as a thermistor or they may be complex semi-intelligent devices with high-speed digital interfaces, such as a digital camera. For the latter case the EvBot should be able to support as many digital interfaces as possible. The process of adding and removing components should be as “Plug-and-Play” (PnP) as possible; if new drivers must be written, the interface should be as simple and flexible as possible.

The EvBot must be able to provide a wealth of data. The onboard processing power and battery life should be maximized while maintaining the small size and low cost described previously. Not only must the EvBot be capable of interfacing with smart and passive onboard sensors, but it must also be able to gather data from remote sensors, whether on or off of another EvBot. By utilizing their communications abilities, EvBots should be able to share information, including raw and processed sensor readings; an individual EvBot should be able to both request and volunteer such information. The EvBot should also be configurable to poll pre-distributed smart sensors and acquire data from them, as would be required to interface with the Smart Dust project [33] or Cricket-based devices [34]. An experimenter should be able to collect all of the data gathered by an EvBot, in both raw and processed forms, including live video. This degree of information sharing, especially when sound and images are transmitted, requires a high-bandwidth communication network.

As a final rule, the EvBot must be a user-friendly development platform. There should be a rich development environment available for the EvBot. C/C++ compilers are a necessity, but to speed development of new experiments the EvBot should also be able to locally run complex controllers written in much higher-level languages. The ability to simulate an EvBot easily and quickly is also an important feature, since even moderately complex evolved controllers would take years to train on a real EvBot.

Section 4.2 Derived EvBot Requirements

Were it not for the size and cost requirements, the other fundamental requirements could be met with little difficulty using readily available parts, as existing robots have already done [30][32]. It is difficult to package a powerful CPU, high bandwidth communications, and support for advanced sensing capabilities into a small, inexpensive robot, especially while maintaining ease of software development.

To enable the most rapid software development, an unrestricted very-high-level programming language with simulation capabilities for controller development must be supported by the EvBot. MATLAB is such a language. It is well understood by the international research community, and its ability to dynamically link with C-code fully supplements its extensive native capabilities. Although expensive tools are available to compile most MATLAB-code into C-code capable of running on many platforms when linked with a special MATLAB library, the preferred means of running MATLAB controllers is within the MATLAB environment (hereafter referred to as just “MATLAB”). The use of MATLAB would allow easy debugging and modification of MATLAB controllers installed on an EvBot as they run, and it would simplify the

process of adding a new MATLAB controller to an EvBot—just copy the MATLAB-code M-files implementing the controller onto the EvBot. MATLAB would also enable easy simulation of EvBot controllers, since the same controller code that runs on the EvBot can also run on a high-powered desktop workstation. By providing a MATLAB-accessible simulation of the real world (itself possibly coded in MATLAB) to a copy of MATLAB running on a workstation, a MATLAB-code EvBot controller could be fully trained in simulation and then be directly transferred to an EvBot for real-world experimentation. Because of all its advantages, MATLAB support was deemed necessary for the EvBot.

High-bandwidth communications can be achieved through various means, but radio-frequency (RF) communications are required when wireless connectivity must be maintained without line-of-sight paths between EvBots. Because most multi-agent RF communication links share a fixed amount of bandwidth between all participants, such as the entire colony of EvBots, the potential for short range inter-robot communications is also required. Short range communications can keep the number of EvBots sharing a single communications link small, maximizing the bandwidth available to each EvBot. The two most popular protocols for wireless RF communication are wireless Ethernet and Bluetooth. Both supply sufficiently high-bandwidth to allow a remote computer to control an EvBot, were that necessary. Therefore, the EvBot should support one, if not both, of these protocols.

The CPU on a EvBot must be able to run complex controllers. Because these controllers may be written in MATLAB—and as such require MATLAB to run—the

CPU must actually be able to run MATLAB itself and still have sufficient power to run the user's controller. MATLAB is currently only available for PC-compatible hardware and high-powered workstations, effectively constraining the EvBot's CPU architecture to PC-compatible.

Software on the EvBot must be able to surrender partial and/or complete control to a remote computer. It must also provide as simple of a device-driver interface as possible without sacrificing flexibility. Software protocols are necessary to enable unrestricted information sharing between EvBots, and support for distributed smart sensors should be easy to add. A means of data logging to other machines is another requirement. The EvBot's operating system must be supported by MATLAB (Linux or Windows), and numerous development environments, including C/C++, should be available in addition to MATLAB.

The use of MATLAB also dictates other design choices. At least 16 MB RAM and 60 MB of nonvolatile storage must be available to run MATLAB, and additional RAM and storage will be required by the operating system and the user's controllers. To achieve the flexible and robust sensor interfacing required to provide a wealth of data on a PC-compatible system, all the standard peripheral interfaces (serial, parallel, and USB) must be available, in addition to general-purpose analog and digital inputs/outputs as are common on virtually all robots. Combining these requirements with those above, the following minimal hardware is necessary to meet all the fundamental requirements:

- Wireless Ethernet or Bluetooth
- Pentium class CPU

- Chipset and associated hardware supporting PnP modular connectivity
- 32 MB RAM
- 80 MB nonvolatile storage
- All standard computer interfaces (serial, parallel, and especially USB)
- General purpose analog and digital inputs and outputs

The following software and software capabilities are also necessary to meet all the fundamental requirements:

- MATLAB
- MATLAB supported operating system (Linux or Windows)
- Numerous development environments, especially C/C++
- Simple but flexible device driver interface
- Remote data logging
- Communication protocol(s) for EvBots
- Support for an interface protocol for distributed smart sensors
- Remote control interface

To better meet anticipated future requirements, some of these requirements were raised for the EvBot; the required RAM was increased to 64 MB and the storage to 104 MB. Ethernet was chosen over Bluetooth (and other technologies) as the wireless RF interface of choice due to its market share, ease of communication with existing computers [27], and ability to work in broken Ad Hoc networks [35] (although USB connectivity should allow the EvBot to be easily equipped with short-range Bluetooth RF communications in the future). USB support is especially important for both its

versatility and compactness. Numerous devices can be connected to a single USB interface using one or more compact USB hubs (shown beside the EvBot's camera in Figure 4.1), and simple development kits are available to interface custom sensors to USB (<http://www.cypress.com/products/>). An onboard USB web cam digital camera was chosen to be the first supported complex sensor due to its versatility and ease of connection. All of the EvBot's equipment should draw as little power as possible, and it should cost less than \$1000, while fitting within a nine inch diameter circle.



Figure 4.1 Compact USB hub shown beside the EvBot's USB camera for size comparison.

Section 4.3 Hardware Design

The EvBot was designed as three hardware subsystems: the TB, the utility board (UB), and the PC/104 stack, each shown in Figure 4.2. Figure 4.3 shows an actual photo of an EvBot. The PC-compatible PC/104 stack normally acts as the brain of the system and is capable of running MATLAB locally while communicating via wireless Ethernet. The TB with its onboard MCU is typically slaved to the PC/104 stack; it is responsible for EvBot locomotion, device actuation, and sensor interfacing for simple, passive

sensors. In many ways, it is similar to the JE. The UB physically connects the PC/104 stack to the TB and provides the PC/104 stack with utility functions. Complete construction details for all three subsystems can be found in Section 8.2; these systems are described below.

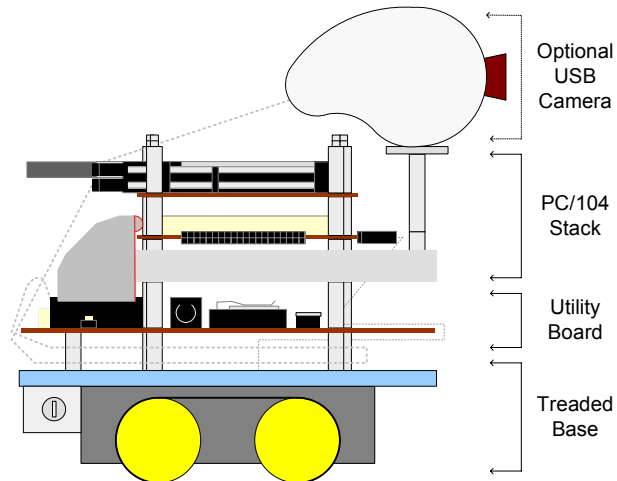


Figure 4.2 EvBot hardware diagram

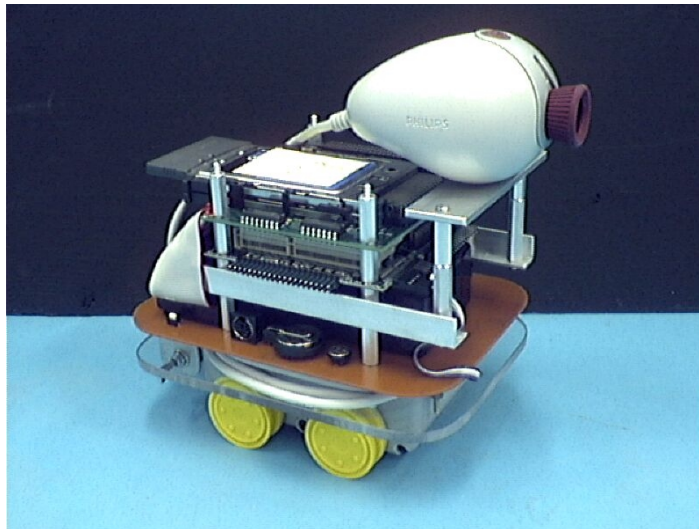


Figure 4.3 EvBot equipped with camera and with treads removed

Section 4.3.1 PC104 Stack

To meet all the hardware requirements, the embedded systems market was explored and a solution was found in the modular PC/104 embedded computer standard, used in both [27] and [28]. PC/104 is a form factor and bus layout for small, stackable modules compatible with the PC standard. Each PC/104 module, whether a motherboard or PWM controller, measures 3.6 x 3.8 inches, well within the 9 inch boundary of the EvBot. Each module is connected to the other modules physically by stacking and electronically via a special stacking version of the ISA bus, preventing significant space from being consumed by the addition of modules (see Figure 4.4). While the ISA bus is sufficient to interface with most peripherals, there is also a PCI compatible version of PC/104 (named PC/104+) that is physically compatible with the PC/104 standard save for the addition of a stacking PCI bus, enabling PC/104 and PC/104+ components to interoperate when stacked together.

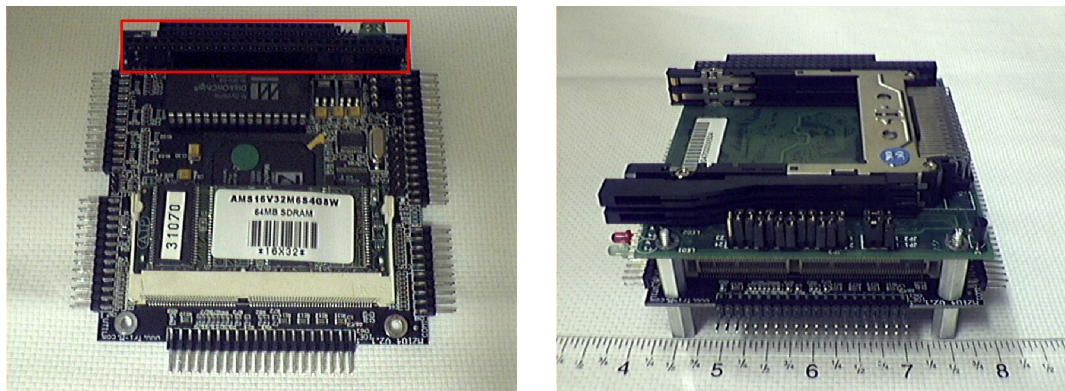


Figure 4.4 PC/104 modules connect electronically using a stacking version of the ISA bus, boxed in red on the left, and mechanically using 0.625 inch standoffs, shown on the right.

Section 4.3.1.1 CPU, RAM, and I/O

To minimize the size and cost of the EvBot, as many functions as economically possible must be integrated into each PC/104 module. Tri-M's MZ104 was chosen for the PC/104 motherboard (shown on the left in Figure 4.4). Like many other PC/104 motherboards, it has integrated keyboard, mouse, floppy, hard disk, serial, and parallel interfaces; it includes a Pentium/586 class CPU and can support 64 MB or more of SDRAM. It separates itself from the others, however, by also including USB support and by drawing the least amount of power for a Pentium/586 class motherboard. As a PC/104 motherboard, it does not support the PCI bus, but in the future it could be replaced in an EvBot's PC/104 stack with a PC/104+ motherboard, were the high throughput of PCI necessary. It achieves all the specifications mentioned by using the ZFx86 fully integrated system-on-a-chip from ZF Micro Devices (<http://www.zfmicro.com/>). The ZFx86 includes other necessary PC support hardware too, such as an embedded PC BIOS, a PnP motherboard chipset, and a fail-safe Boot ROM. For the EvBot, the ZFx86 CPU core was configured to run at 100 MHz, the highest user-selectable frequency at which it would run stably. The MZ104 can even power the entire PC/104 stack when fed a single +5V input; it includes the DC-DC converters necessary to supply $\pm 12V$ and $-5V$ from the +5V supply. The only standard PC component missing from the MZ104 is a video interface, which is unnecessary for the EvBot during operation and would thus waste valuable power. (During EvBot development or maintenance, a PC/104 video card can be added on top of an EvBot's PC/104 stack, turning an EvBot into a complete PC system.)

The MZ104 also includes a DiskOnChip (DOC) 32-pin DIP socket supporting both the DOC 2000 and the DOC Millennium. The DOC is a compact, nonvolatile flash disk which automatically loads BIOS extensions to make itself appear as a normal, *bootable* hard disk to the BIOS. It also features patented wear-leveling technology to prolong its life through thousands of write operations, even if all those writes are to the *same logical sector* of the DOC. The EvBot was equipped with an 8 MB DOC, responsible for booting the EvBot and containing the core operating system.

Another simple but very important feature of the MZ104 is that it uses standard headers with 0.1 inch pin spacing as the physical connection for all of its interfaces. Other PC/104 motherboards combine several interfaces into single high-density connectors; doing so on the EvBot would increase cost, complicate assembly, and make adding or removing individual devices difficult. EvBot bulk was further reduced by using 0.1 inch headers where appropriate at the other end of the interfaces, rather than the much larger connectors normally used for each interface.

The EvBot also includes a PC/104 PC-Card interface module (also known as a 16-bit PCMCIA interface module). The PC-Card interface is used to connect up to two PC-Card devices to the PC/104 ISA bus. It is shown on the right in Figure 4.4 where it is the top module of the PC/104 stack. Two PC-Cards are required for a normal EvBot: one for most of the EvBot's flash storage and another for wireless Ethernet connectivity.

Section 4.3.1.2 Flash Storage

The EvBot uses a 96 MB ATA flash card to store its controller code, in addition to any execution environment (such as MATLAB) and support programs needed to run

that controller. The card has a PC-Card interface and form-factor, making it very compact, consuming virtually no extra space when installed in the PC/104 PC-Card interface. As a flash device, it is much more compact, shock-resistant, and power-efficient than a hard disk. It is also slower and more expensive, though, and so only 96 MB are used. The ATA flash card standard virtually guarantees software compatibility across manufacturers by mimicking a normal ATA hard disk interface; however, the interface is only available to software after the PC-Card interface is initialized, which normally prevents booting from an ATA flash PC-Card. Because the ATA flash card is a PC-Card device, it can be easily removed from an EvBot and plugged into a Laptop, providing the user with one method to modify an EvBot's controller. Using a wireless network connection to modify the controller in place on the EvBot is another.

Section 4.3.1.3 Wireless Ethernet

The EvBot's other PC-Card is a wireless Ethernet interface. Like the ATA flash card, it consumes little space when installed in the PC-Card interface (although it does have a compact protruding antenna). To maximize RF signal strength, the wireless Ethernet PC-Card should be inserted above the ATA flash card in the PC-Card interface, and the PC-Card interface should be installed above the MZ104 motherboard, ideally on top of the EvBot.

Section 4.3.1.4 USB Web Cam

Although not an integral part of the EvBot, a USB web cam digital camera was the first complex sensor installed on the EvBot. When installed, the camera is mounted

in a fixed position to the front of the EvBot above the PC/104 stack (because the wireless Ethernet antenna is at the back of the EvBot it remains unobstructed). The camera chosen was the Philips Vesta Pro Scan. Not only does it have hardware support for VGA 640 x 480 resolution, but it also has an integrated USB microphone with an attached 44.1 kHz analog-to-digital converter (ADC). Both of these devices share the same USB connector and are fully supported by both Linux and Windows. Other features of the Vesta Pro Scan include its excellent low light sensitivity and its high image quality for an inexpensive (\$50) camera.

Section 4.3.2 Treaded Base

The TB was designed to be as generic as possible, even useful without the PC/104 stack. It is capable of at least three modes of operation. First and foremost, the TB can be wired to the PC/104 stack to be given commands by the MZ104, as it is when used with the EvBot. Second, with the addition of a RS232 wireless link and a minor firmware change, the TB could be controlled wirelessly by a remote computer using almost the same protocol as is used by the MZ104 to command it when it is physically wired to the PC/104 stack. Third, by replacing the firmware altogether, the TB could even be made to execute simple controllers locally for autonomous operation. However, because the TB was developed primarily for use with the EvBot, firmware was only written for the first configuration. Therefore, details will only be provided on the use of the TB for the that purpose.

The core of the treaded base is the same skid-steered, tread-drive section of a toy vehicle used for the JE (Section 3.1.1). Like the JE, the TB includes H-bridge motor

drivers and an MCU capable of general purpose analog and digital I/O. Although not currently implemented in the EvBot design, the TB could easily be connected to an additional wireless link using one of the TB's RS232 interfaces; this could provide an EvBot with a useful alternative to Bluetooth for short-range low-bandwidth communications. When connected to the PC/104 stack, the TB's MCU interfaces with the MZ104 motherboard using a standard RS232 serial link connected to the primary serial port on each device. Unlike the JE, the TB does not use a proto board to hold its components. Rather, the MCU and H-bridges, which are more advanced than those in the JE, are mounted on a custom PCB located within a small hollow area in the top of the tread-drive unit, as shown in Figure 4.5 and Figure 4.6.

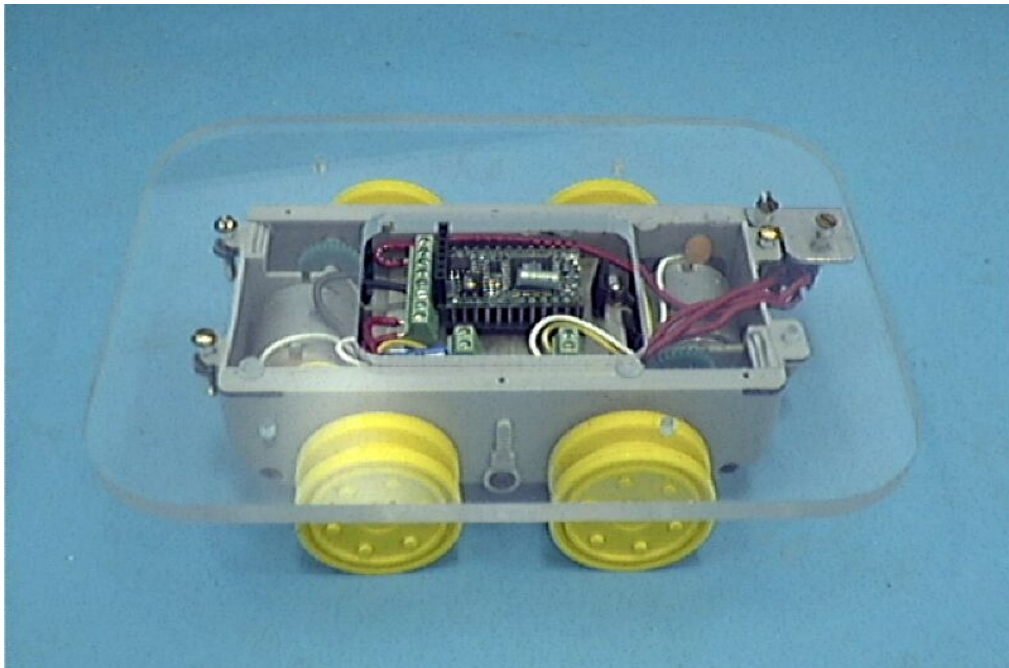


Figure 4.5 The EvBot's Treaded Base

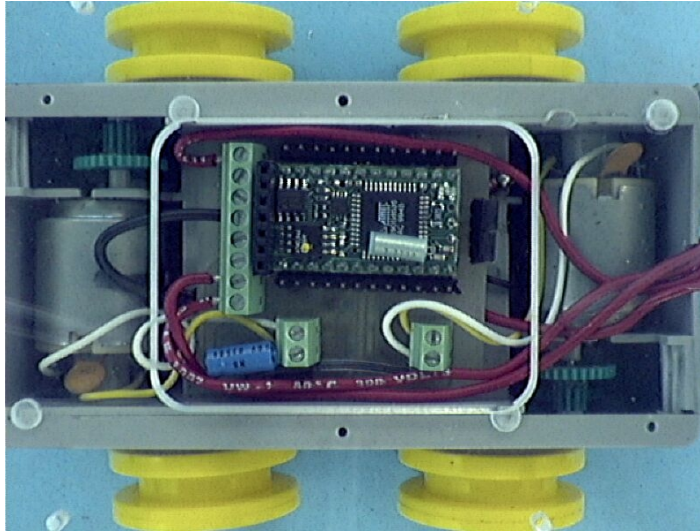


Figure 4.6 Treaded Base close-up

Section 4.3.2.1 BasicX MCU

The TB uses NedMedia's BasicX-24 (<http://www.basicx.com/>) as its MCU. The BasicX-24 (henceforth referred to as the BasicX) is pin compatible with the BS2 used on the JE (Section 3.1.1.1). It is a low-power device capable of duplicating virtually any BS2 functionality. However, the BasicX has the following important features lacking in a BS2:

- Speed: roughly 16 times faster than the BS2
- Full multitasking support
- Real interrupts, with one external interrupt request pin
- Interrupt driven I/O on the first RS232 port's UART
- Dual PWM generators
- Integrated ADC
- 400 bytes RAM

- 32 KB EEPROM for user code (the TB's firmware)
- Built-in feature-rich operating system
- Full IEEE floating point math
- Pin-less connections for additional I/O ports (beyond 16), including SPI

The first five of these features are necessary to effectively implement a closed-loop wheel control system that would be capable of maintaining a fixed velocity and direction while responding to commands from the PC/104 stack. The ADC provides a means to read analog voltages from simple passive sensors, complementing the digital I/O capabilities present on all 16 general purpose pins. The increased RAM and EEPROM, together with the TB's operating system, enable more complex firmware to be written to take advantage of the BasicX's additional capabilities. Finally, the floating point math support and additional I/O connections add robustness as they may be important features for future additions to the EvBot's TB.

Section 4.3.2.2 Efficient H-Bridge PWM drivers

The H-bridge motor drivers used on the TB are made of very efficient high-current inverters, as shown in Figure 4.7. One inverter is connected to each pole of each DC motor. The inverters basically function as high current buffers; their inverting nature is compensated for in the TB's firmware. When the MCU wants to spin a motor in one direction, it figures which pole should be at high voltage and which should be grounded. It then outputs the opposite logic values to the inputs of each pole's inverter, causing one inverter to supply current to one of the motor's poles at supply voltage and the other inverter to sink current from the motor's other pole to ground. The inverters also allow

the motors to be supplied from a different power source than the rest of the EvBot, keeping motor noise from interfering with the PC/104 stack. Furthermore, the MCU has two PWM outputs that can each be routed to one of the inverter inputs for each motor, allowing full, independent speed and direction control for each tread. Actually, an EvBot does not need to use treads; both drive wheels for a given tread are linked with gears to the motor. If treads are not needed for traction they can be removed to reduce friction when turning.

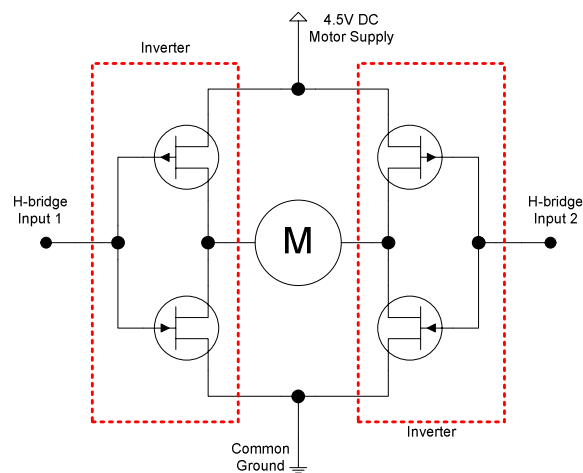


Figure 4.7 CMOS inverter-based high-efficiency H-bridge

Section 4.3.2.3 Encoder Support

Although not currently implemented, the BasicX and connected H-bridges provide all the necessary hardware required to implement an encoder-based full-feedback velocity control system for each tread. (Such an implementation is currently underway at the CRIM.) When connected to encoders, the BasicX could actively regulate motor power to maintain a desired EvBot speed and direction. A schematic for sharing the

BasicX's single interrupt request line between the two encoders is provided in Appendix Section 8.2.1.1.

Section 4.3.3 Utility Board

When the TB is used as part of an EvBot, the TB has a UB fastened to the top of it. The UB is responsible for supporting the PC/104 stack, both physically and electronically.

Section 4.3.3.1 Power Supply and Battery

The most important electronic component of the UB is its power supply. The UB carries a 5 V regulator capable of supplying power to the PC/104 stack via the MZ104. The regulator is a low-drop-out (LDO) regulator with built-in low-voltage detection. If its output voltage drops below a threshold, it continually resets the PC/104 stack to prevent the flash disks from being damaged by operating with insufficient voltage. The UB also carries the battery that powers the regulator, typically a 7.2 V 3000 mAh Ni-MH battery capable of powering the EvBot's electronics for over two and a half hours (the TB's motors are powered by the TB's own batteries, which last even longer).

Section 4.3.3.2 Utility Devices and Interface

The UB is also responsible for connecting several components to the MZ104 motherboard. It contains the real-time-clock (RTC) battery that powers part of the MZ104 when an EvBot is turned off, allowing the MZ104 to keep time and remember its CMOS settings. It holds a miniature speaker connected to the MZ104 that beeps like a normal PC speaker; such functionality is very useful when debugging either an EvBot or

a new controller. Also on the UB are connectors for keyboard and USB. All of these components connect to the MZ104 through a single cable.

Section 4.3.4 Support Hardware

To effectively experiment with an EvBot colony, some hardware is needed in addition to the EvBots. A wireless Ethernet access point improves wireless communications efficiency by acting as a “traffic cop.” It can also bridge the wireless network to an existing wired Ethernet infrastructure, allowing a hard-wired computer to communicate directly with an EvBot. If the access point is connected to the Internet, it is even possible for an EvBot to communicate with computers around the world, possibly allowing multiple laboratories to share a single EvBot colony. EvBots are also much easier to maintain when they are automatically assigned their network IP address and machine name each time they bootup; this can be accomplished by having a server connected to the EvBot network that runs a DHCP server. That same computer can also act as a central place for the EvBots to log data and save sensor readings for future review. Essential auxiliary items include a keyboard, mouse, monitor, PC/104 video card, battery chargers, and an adjustable voltage power supply to eliminate battery usage during long periods of EvBot development. More details on implementing several aspects of a complete EvBot support environment are provided in Section 8.4.

Section 4.4 Software Design

The EvBot’s software works with its hardware to meet the fundamental design requirements. Communication between the PC/104 stack and the TB follows a protocol.

Device support is relatively easy to add. A small Linux distribution was created that is capable of running and supporting MATLAB controllers on the EvBot's minimal hardware.

Section 4.4.1 Treaded Base Interface Protocol

The TB interface protocol (TBIP) is used by the PC/104 stack to communicate with the TB. It is the language spoken by the TB firmware and the Linux TB-interface system. Since the TB's MCU is responsible for overseeing EvBot locomotion, actuating users' devices, and reading their simple sensors, it is impossible to predict every type of device that the TBIP may need to control. To ensure robustness with respect to future EvBot capabilities, the TBIP must be extensible by the user to handle their unique requirements. Therefore, the protocol was designed to be as simple as possible while maintaining flexibility and extensibility. A command-response paradigm was chosen for the TBIP, with the structure of each command regulated only enough to insure the interoperability of commands.

Each command begins with a single one-byte character that identifies the exact command being transmitted, which determines the number and type of arguments that will follow. After receiving and executing a command, the TB always responds by echoing back the command's identification character followed by any data that needs to be returned. Were the TB's MCU to fail in executing a command (possibly due to noise or physical obstruction), the TB instead transmits an error message, allowing the stack to either retransmit the same command or send a new one.

Section 4.4.2 Method for Adding a New Device

When a user adds a new device to the TB, they must extend the TBIP to include a new command and thus modify both the TB firmware and Linux TB-interface system. New devices connected directly to the PC/104 stack, on the other hand, can be supported just as they would on a desktop computer. If the device was purchased, then instructions for using it with Linux may be documented by the device's manufacturer or be available on an appropriate Linux-support website. If not, or if the device was custom built, then a Linux driver will have to be written for it. Numerous books and articles are available on the subject of writing Linux device drivers, and since there are no special considerations for writing Linux device drivers specifically for use on the EvBot, nothing more will be said about the process.

Section 4.4.2.1 TB Firmware Modification

Extending the TBIP requires modification of the TB firmware. Since the TB uses a BasicX as its MCU, the firmware is written in the Basic Express language (BasicX code). Basic Express is an enhanced subset of Visual Basic that allows rapid coding of support for new devices as they are added to the TB. Although the current TB firmware does not use multitasking interrupt handling, it is structured to allow such, as would be required for onboard, closed loop velocity control. At present, it implements a few essential commands, all of which perform a short task and then return. As such, these commands are completely contained (and serialized) within the main loop that handles the TBIP. To enable multitasking interrupt handling, the user would need to add an

interrupt handler (and possibly other tasks) that runs concurrently with the main loop. The Basic Express language makes this easy.

The main loop iteratively receives and processes commands from the PC/104 stack. It is here in the TB's firmware that support for a new device must be added. Command parsing is handled with a finite state machine (FSM) that has one state for the processing of each command. To add a new command for a new device, the user needs only to add a state to the FSM. That state can contain any code the user deems necessary, but must handle the reading and parsing of any arguments for that command. If the command requires that some action be continually and periodically taken thereafter, it can start (or control) a separate, concurrently running task using the BasicX's multitasking capabilities. Several helper functions are also included for convenience in the existing TB firmware. Complete BasicX code implementing the current EvBot's TB firmware (and thus the TBIP) is available and further described in Section 8.3.1.

Section 4.4.2.2 Linux TB-Interface System Modification

TBIP modifications to the TB firmware must be matched in the Linux TB-interface system to be useful. The Linux TB-interface consists of a server program that talks directly to the TB and numerous client programs that can be called by a controller. There is typically one client program for each command of the TBIP. All that is needed to make the Linux TB-interface support a new command is to write a new client program that sends the command to the TB through the server program and processes the TB's response, which is received from the server program. The short C-code for the existing

client programs (listed in Section 8.3.2.6) can be used as a simple template for quickly coding new client programs.

Section 4.4.3 Linux Distribution

Section 4.4.3.1 Need for Linux

As stated previously, either Linux or Windows is required to run MATLAB on an EvBot. However, an EvBot has only 104 MB of nonvolatile storage, and a minimal install of MATLAB uses at least 45 MB of storage, leaving roughly 60 MB free. While some additional space can be made available by judiciously deleting MATLAB documentation files and support utilities, an additional 20 or 30 MB can be quickly consumed if MATLAB toolboxes are needed by a controller. Thus, if several toolboxes are needed, then only 30 MB may be free for both the operating system and the controller code. It is impossible to install a 32-bit version of Windows on just 30 MB of storage, but Linux can be installed on just a few MB by removing all but its most essential components. Therefore, Linux is the only operating system capable of supporting and coexisting with MATLAB on just 104 MB of storage, making Linux a requirement for the EvBot.

Section 4.4.3.2 Nature of Linux

Linux is an advanced open-source UNIX-like operating system available for numerous hardware architectures. Because it is open source, those with adequate time and skill can modify any aspect of it to fit their needs, ensuring the possibility of device-driver support for any hardware component a user may add. Dedicated users around the

world help maintain Linux, eliminating bugs and adding support for new hardware. Due largely to their efforts, Linux is probably the most robust operating system currently available for the PC platform. The core of Linux is the kernel and its associated kernel modules. The kernel is the first part of Linux to load when the computer boots. It always runs while the computer is in operation, policing and providing services to all other running programs. The kernel alone is able to talk directly to hardware, and so device drivers must either be compiled directly into the kernel or loaded into the kernel as modules while the kernel is running. At the time of writing, Linux supports more x86 (PC-compatible) hardware than any other non-Microsoft operating system. The kernel itself does not interact directly with the user, and so to make a “normal” operating system out of Linux (referred to as a “Linux distribution”), the Linux kernel must be combined with many libraries, system utilities, and configuration files which build upon the kernel’s functionality to provide a convenient, feature-rich environment for starting and running programs.

Section 4.4.3.3 The Infinite Atom Linux Distribution

Most Linux distributions are large, consuming anywhere from several hundred to a few thousand MB of storage space. Most of this space is used for a selection of graphical user interfaces (GUIs) and hundreds of applications bundled with the operating system which are unnecessary to run MATLAB or operate the EvBot. Because storage space is scarce on the EvBot, a small yet powerful custom Linux distribution (paradoxically named Infinite Atom) was built for the EvBot to run MATLAB and yet fit entirely on the 8 MB DOC, leaving the 96 MB ATA Flash Card free to store MATLAB

and the user's programs (whether they be MATLAB controllers or additional Linux executables). This dichotomy of storage space helps protect the Linux distribution, which is more complex than MATLAB and likely to be less understood by the user than either MATLAB or the user's own additions to the EvBot's software. This protection is necessary to ensure robust stability because the Linux distribution is critical to the EvBot's operation. Additional protection of the Linux distribution is provided by keeping the DOC mounted read-only, allowing only "super-user" certified, hopefully competent individuals to change it, as would be required for an EvBot upgrade.

Actually, both the DOC and the ATA Flash card are normally mounted read-only to prevent data corruption should the battery die, but normal users can easily change the flash card to and from writeable mode as necessary to modify their controllers. Keeping all the nonvolatile storage mounted read-only poses a few potential problems, most of which were circumvented by using 4 MB of RAM as a third, writeable storage device called a RAM disk. Not only does the use of a RAM disk increase robustness, but it also speeds up file writes significantly (flash disks are even slower than hard disks, while RAM is always much faster).

A significant part of most Linux distributions is devoted to a plethora of software development tools, including compilers (C, C++, and JAVA) and entire programming environments. There is no need to place these tools on an EvBot since they can be used much more easily on a desktop computer, and the binaries they produce there can be easily transferred to an EvBot where they run without modification. Hence, the entire set

of easy-to-use development tools for Linux is effectively available for EvBot development.

Like most distributions, Infinite Atom saves space by dynamically linking most libraries with executables at runtime. One of the most important features of the Infinite Atom distribution, therefore, is that it provides all of the libraries needed to run both the EvBot's software as well as most programs a user might want to add to the EvBot. To help ensure all necessary libraries (and other files) were included, several other small Linux distributions were consulted, some of which use only a few MB—such as Tom's Root/Boot Disk (<http://www.toms.net/rb/>)—and others which use closer to 100 MB—such as Linux Care's Bootable Recovery Disk CD-ROM (<http://lbt.linuxcare.com/>). While these distributions, and several others, were consulted to see which files to include, the actual files were taken almost exclusively from beta and final versions of Red Hat 7.1 and 7.2, to ensure full compatibility with real applications. Before a file containing executable code would be placed on the EvBot, however, it would first be stripped to free any space consumed by object file symbols. Despite the careful selection of included libraries, a user may still want to add additional special-purpose libraries to an EvBot. Provision for such libraries has been made; they can be placed in a designated directory (lib/) on the ATA Flash Card where Linux will automatically find them when the programs that need them execute.

Section 4.4.3.3.1 Full Wireless Ethernet

Another important feature of Infinite Atom is that it contains the necessary utilities, configuration files, and server processes to fully support wireless Ethernet

networking. Programs communicate over the EvBot's wireless network just as they would over a wired network in an office building, providing an easy means of communication with an EvBot. It is simple to setup remote data logging and/or remote control of an EvBot, using either standard network communication tools or custom programs written to fit the specific needs of a user. Typically, an EvBot will be equipped with secure shell (SSH) on its flash card, a standard network communications program enabling users to securely log into an EvBot to check its status, change its operating parameters, or assume complete control of it. Wireless Ethernet also provides an ideal means for EvBots to communicate with each other. Its high bandwidth can handle a constant stream of raw and processed images being transferred between EvBots, or even a few simultaneous high-quality live video transmissions. The only remote devices for which wireless Ethernet may not be well suited to communicate are those that are too simple to support it, such as distributed smart sensors. For these devices the TB can be used to add an additional simple wireless communication link to the EvBot, and the TBIP can be expanded as outlined above to support it.

Section 4.4.3.3.2 Complete Imaging System

New to the latest series of Linux kernels is high-quality support for USB web cams. The best supported low-cost web cams are made by Philips, and so the necessary drivers (kernel modules) for Philips web cams are included in Infinite Atom as well. Unfortunately, the newest 2.4.x series of kernels also has serious difficulties booting properly from the DOC of an EvBot. Full details of the process underwent to make Infinite Atom boot from a DOC are included in Section 8.3.2.1.

Section 4.4.3.3.3 Full MATLAB Support

The most challenging task for Infinite Atom is to provide an environment capable of running MATLAB and supplying it with everything it needs to completely control the EvBot, including even the ability to communicate with other EvBots. Fortunately, MATLAB has the ability to execute other applications and read their return values, as well as the ability to perform file I/O. Combined, these mechanisms provide a sufficient interface to empower MATLAB with complete EvBot control.

The Linux TB-interface system provides a means for MATLAB to control the EvBot. Each TB-interface client utility program is responsible for executing one command of the TBIP by sending the command over the appropriate serial link and reading back the response; the utilities take command line arguments and return their values by printing them to the screen. Because a MATLAB controller may call several TB interface utilities in rapid succession, and because Linux does not work properly if a serial port is opened and closed faster than once a second, a separate program (the Linux TB-interface server) was necessary to open the TB-interface serial port and keep it open as long as the MATLAB controller is running. This BasicX server uses named pipes to relay data back and forth between the TB interface utilities and the serial port connected to the BasicX (which is designated by a symbolic link).

MATLAB acquires images from the EvBot's web cam (if it is present) by using a normal command-line Linux program named `vgrabj` to capture an image from the camera and save it to a file on the RAM disk. Each time `vgrabj` completes, MATLAB reads the image file it saved and processes it to see where the EvBot is going. If

MATLAB detects that the camera's video settings have been altered somehow, it can use another Linux program named `setv4l` to restore the settings to their desired values.

Currently, MATLAB's only method of direct communication with other EvBots is with the HTTP protocol used on the web and supported by Linux. Bundled with Infinite Atom is `thttpd`, a small, simple, and very fast web server. By including `thttpd` on the flash card and starting it automatically on bootup, each EvBot becomes its own web server. MATLAB can then write to the files served by the web server in order to communicate anything it wants to the world (or at least to any other machine connected to the EvBot network). EvBots can then use the `w3c` command-line tool to fetch the web pages of other EvBots and read what MATLAB is saying on them. As long as every EvBot periodically checks the web pages of the other EvBots, an EvBot can share important information with any other EvBot, or request that one or more EvBots provide it with necessary data. The `w3c` program also allows the EvBots to upload images and other data to a remote server for convenient review at a later date.

MATLAB has also been trimmed down as much as possible to maximize the nonvolatile storage available for the user's controllers and necessary toolboxes. Documentation was removed from MATLAB, as were some installation artifacts not actually used during operation. Some 3D graphing and printing support was removed as well (EvBots lack both attached displays and printers, and drawing or printing graphs is useless for controlling robots).

For historical compatibility reasons, the older 5.x versions of MATLAB used on the EvBot ship with their own versions of several Linux libraries. These libraries had to

be installed on the ATA Flash Card for MATLAB to operate properly. To insure that all the rest of MATLAB's requirements were met by Infinite Atom, the distribution was booted and MATLAB loaded several times; each time that MATLAB would request an uncommonly used system-utility not already included in Infinite Atom, the utility would be added to the distribution and MATLAB would be restarted.

As configured by default, Infinite Atom automatically starts MATLAB as the last program on bootup by registering MATLAB as a systems-critical program. Should MATLAB ever exit, naturally or unnaturally, Infinite Atom will automatically restart it. MATLAB can be made to automatically run a specific controller as soon as it starts, which provides a means for a MATLAB controller to automatically start controlling an EvBot on power-up.

Chapter 5. Experiments and Results

The key result of this work is that the EvBot worked, and that it met all the design requirements. Without sensors, the EvBot measured only 6.75 x 5 inches. Because of its rounded corners, it could be bounded by a circle of diameter 7.6 inches. The EvBot's hardware cost only \$960, and the EvBot consistently ran for over 2.5 hours using its Ni-MH 3000 mAh battery. The EvBot could communicate with other computers and EvBots, utilize diverse sensors, and run complex MATLAB controllers locally and autonomously. The following experiments illustrate many of these capabilities.

Section 5.1 The Maze Environment

To assist in the creation of useful experiments, a semi-modular closed-maze environment was created for the EvBots, as shown in Figure 5.1. The maze consisted of black wall sections and vertices resting on a monochromatic floor. Each cell of the maze was square, measuring 31 inches on the side. The total maze was also square, measuring five cells on each side. The walls were made a little taller than the height of an EvBot, preventing one EvBot from seeing over a wall to another EvBot. To maintain a consistent environment for consecutive vision-based experiments, a light-colored monochromatic visual barrier was placed around the outside walls of the maze. The barrier helped regulate maze lighting and prevented EvBots from reacting to people in the laboratory. The high contrast between the walls of the maze and their surroundings allowed the EvBots to accurately determine their distances to the walls they could see by

simply counting how many pixels high the walls appeared. This method of distance calculation worked using only a single camera, and it was more accurate at close range.

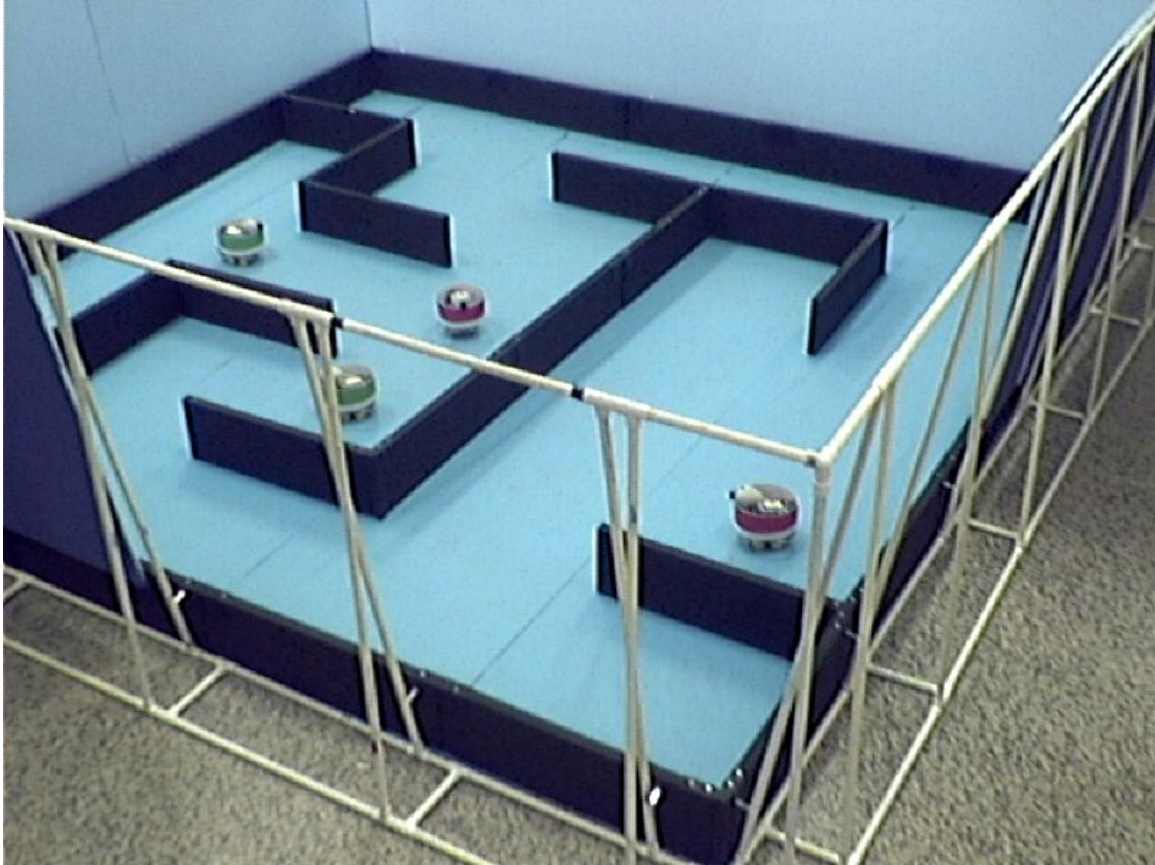


Figure 5.1 The EvBots' maze with some of the visual barrier removed

An overhead camera was mounted above the maze. Equipped with a fisheye lens, the camera was able to see the entire maze environment. It was connected to a computer's video capture card, allowing accurate video and images of the maze environment to be captured during EvBot experiments for quantitative and qualitative analysis. Future plans include the mounting of a gantry above the maze. The gantry would be equipped with an X-Y axis full-motion platform capable of being positioned

over any part of the maze. Cameras and other sensors would be mounted on the platform, which could be used to simulate an unmanned aerial vehicle (UAV) flying over the EvBots below. Other sensors could be directly mounted to the proposed gantry as well. Such a gantry system should also be able to accurately position each EvBot in the maze, possibly using vision-based methods similar to the **iGPS system** described in [36].

Section 5.2 Neural Networks and Touch Sensors

The first experiment to use the EvBot was performed by Ph.D. student Andrew Nelson. He used simulations to genetically train an artificial neural network to control an EvBot equipped only with binary tactile sensors as shown in Figure 5.2. The simulated EvBots were trained to travel as far as possible in multiple simulated mazes within a fixed amount of time. Because the EvBot runs MATLAB, Nelson was able to develop the entire neural network and training simulation quickly in MATLAB, and then easily transfer the trained controller to an EvBot where it operated very successfully in the real maze with virtually no modification. Results are shown in Figure 5.3 which traces the path taken by one of the simulation-trained EvBots as it used an artificial neural network to navigate through the maze. Such transferability from simulation to the real world is often difficult to achieve, and the EvBot's native support of MATLAB programs greatly helped in this case. The ease of controller transfer observed in this experiment also demonstrates a potential for knowledge transfer between multiple EvBots and other forms of shared learning.

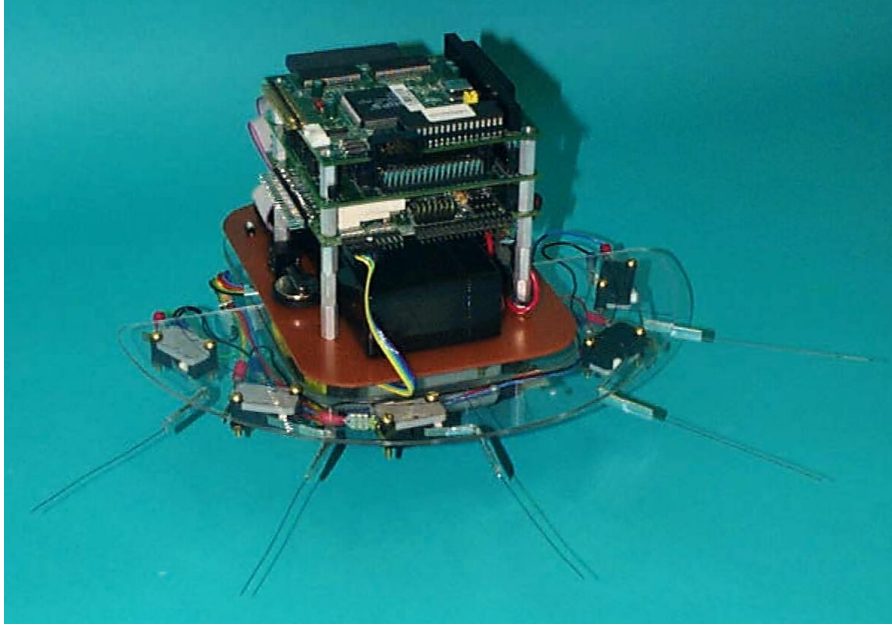


Figure 5.2 EvBot equipped with touch-sensitive whiskers

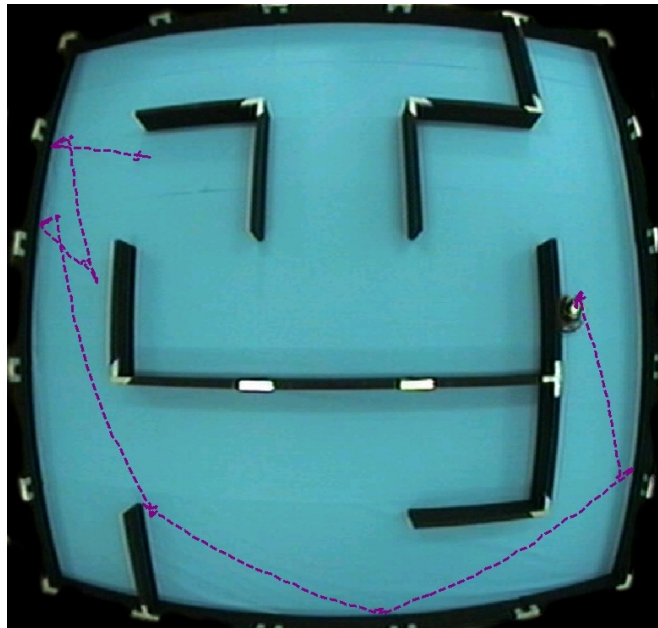


Figure 5.3 Path taken by a genetically trained EvBot (middle right) as it navigated through a maze using only tactile sensors

The experiment showcased the ease with which a user can control an EvBot. Nelson updated the EvBot's controller numerous times by simply copying files over the

wireless Ethernet connection using the standard SSH SCP program. He repeatedly logged onto the EvBot to monitor its status, and occasionally he assumed manual control of the EvBot to explore its behavior when it was placed in certain situations.

As would be expected, however, not all aspects of the EvBot's performance were ideal. By individually polling each tactile sensor attached to the TB, MATLAB experienced a short but considerable delay in sensor reading. The delay was mainly due to task swapping. MATLAB's calling of the TB-interface clients forces Linux to change processes (task swap) several times for each TB command executed. MATLAB does, however, have a facility for linking C-code with a user's MATLAB program, allowing a user to avoid some task swaps by effectively linking the interface utilities directly into MATLAB. This procedure could significantly improve the speed of simple commands that are frequently repeated, and it may be implemented in the future if necessary.

Another deficiency discovered is that EvBot velocity at a given PWM duty cycle varies considerably—and nonlinearly—with PWM peak magnitude, which decays as the TB's batteries discharge. While this is typically not a problem for linear motion, such variance has the potential to introduce real control problems when the EvBot attempts to spin on its axis; what once turned the EvBot 90° may later turn it only 40°. Most controllers should be robust enough to compensate, but some may not be. The ultimate solution is to equip the TB with encoders and have it regulate wheel velocity with a closed-loop control system. Work is currently being done to equip the EvBots accordingly. The last noticeable shortcoming concerns only the tactile sensors that were mounted on the EvBot. Because the sensors were spaced over 1.5 inches apart, it was

possible for the EvBot to run headlong into the edge of a wall-segment and become stuck without triggering any of the tactile sensors as shown in Figure 5.4. This problem could only be solved by using better tactile sensors that have complete coverage around the EvBot. Such sensors have been looked into, but no experiment has necessitated them yet.

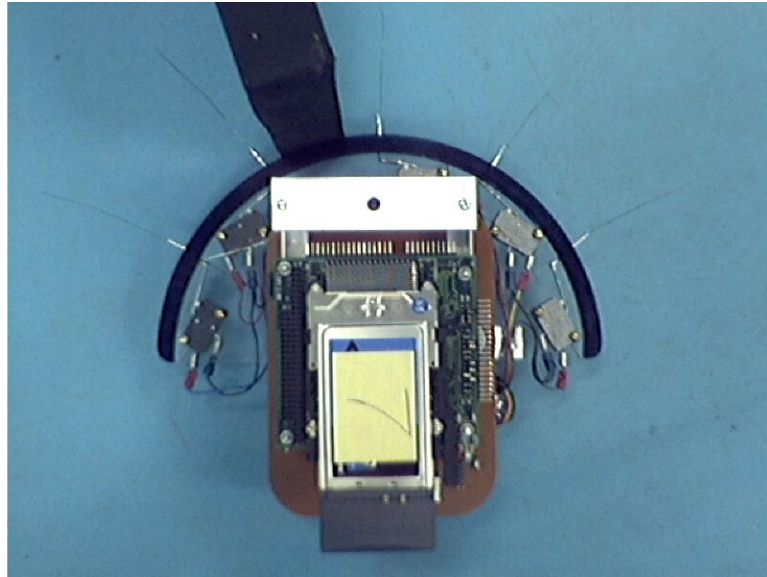


Figure 5.4 An EvBot that bumped into a wall without triggering any of its tactile sensors

Section 5.3 Visually Segregating Robots

Nelson was assisted by the author in setting up a second experimental demonstration to showcase the EvBot's vision capabilities. Four EvBots were divided into two color groups by placing a colored cylindrical shell around each of them as shown in Figure 5.5. Two EvBots were red and two were green. A simple rule-based MATLAB controller was written to make the EvBots seek out other EvBots of their same color in the maze while steering around EvBots of the opposite color. Ultimately, the controller made the EvBots segregate into two groups, one red and one green.

The vision algorithms on the EvBots measure distances by measuring vertical pixel counts of key colors. The colored cylindrical shells placed around each EvBot were of fixed-height, allowing the EvBots to determine EvBot distances using a method similar to that presented in Section 5.1 for wall-distance determination. The EvBots' visual field was divided into several regions to aid in object identification. For example, only the upper half of the visual field was used for wall identification and distance measurement, while EvBot detection and distance measurement were based solely on the lower half of the visual field. This scheme would prevent an EvBot that partially occludes a wall from making that wall look farther away to another EvBot. For this experiment, the EvBots also utilized their data-logging facilities to save all of the images they captured to a server for later analysis. One of the images captured by a red EvBot is provided in Figure 5.6.

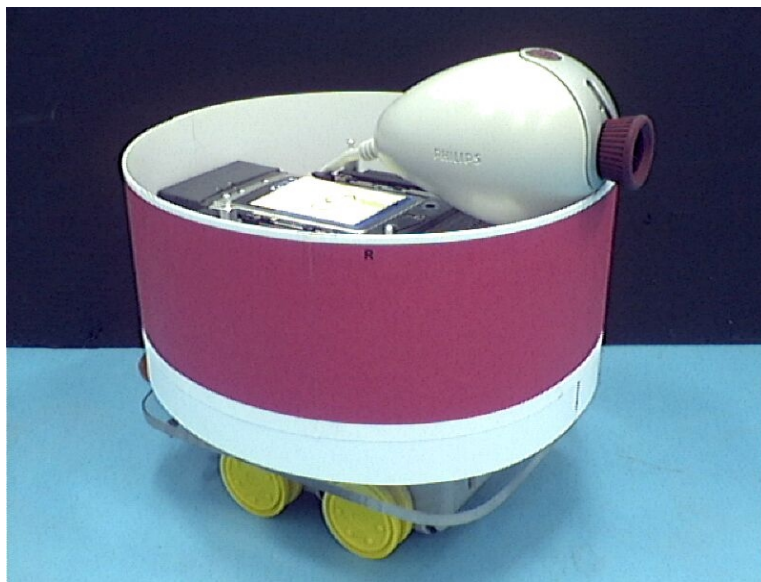


Figure 5.5 EvBot with red cylindrical shell for visual identification by other EvBots



Figure 5.6 This image of an EvBot was taken during an experiment by another EvBot that automatically saved this image to a server for later review and analysis by the human operators.

Once again, minor problems presented themselves. As is the case for most USB cameras, the EvBot's camera automatically adjusts its gain for each individual color when viewing different scenes and in different lighting conditions. This problem was largely solved by a two-fold approach. First, the visual barrier was erected around the maze to control fluctuations in natural lighting. Second, the EvBot controller's color-selection thresholds were carefully set to include the normal ranges of the camera's adjustments. The second problem to present itself was that when equipped only with a camera, the EvBot can get the side of its body stuck on the end of a wall while its camera is looking around the wall and sees no obstructions, as shown in Figure 5.7. The problem was temporarily sidestepped by making MATLAB compare each image to its predecessor. If there was an exact match, the controller would assume the robot was stuck and backup and rotate. A better solution would be to supplement the camera with tactile or proximity sensors that would provide complete coverage around the perimeter of the EvBot. There is currently some interest in adding such sensors for future EvBot-based experiments, but no such work is currently underway. Despite these problems, EvBot visual navigation and interaction was successfully demonstrated.

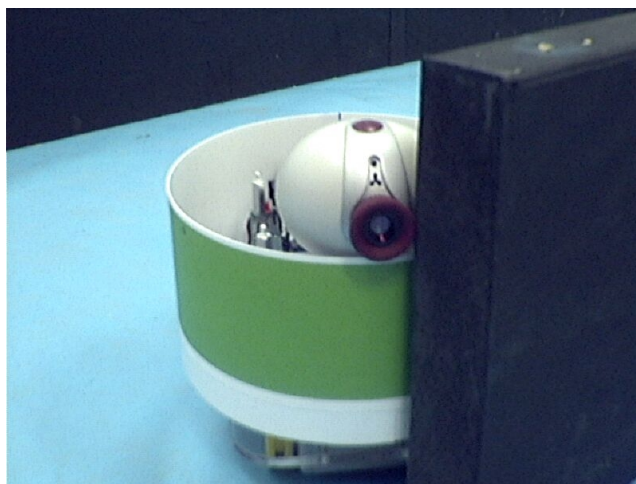


Figure 5.7 An EvBot that cannot see that it is stuck

A “shortcoming” of this demonstration (rather than of the EvBot platform) is that one EvBot can chase another EvBot for an extended period of time, without the leading EvBot ever knowing that it is being followed. For two EvBots to cluster, they must chance to see each other at the same time. Ideally, only one EvBot should need to see the other for them to begin moving toward one another. EvBot communication can make this possible, as the experiments described in Section 5.4 demonstrate.

Section 5.4 EvBots that Communicate

A final set of experiments tested the EvBot’s wireless communication interface for MATLAB. The experiment discussed in Section 5.3 was repeated using two, like-colored EvBots with an alteration to the EvBots’ controllers that allowed the EvBots to share information. If one EvBot were to see the other, it would communicate this. The other EvBot would then stop exploring and start spinning around, looking for the first EvBot. Figure 5.8 shows the paths taken by two EvBots as they traveled through the maze using identical controllers that communicated with one another. Their initial

positions are marked with X's, and they are pictured in their final resting positions where they have clustered together. Note that the EvBot marked by the red (darker) line was headed away from the other EvBot at the top-left corner of the maze, but the EvBot marked by the yellow (lighter) line was able to spot the back of the first EvBot, telling the first EvBot to stop and look around for the second EvBot. Although the EvBots did cluster together at the top right corner of the maze, they did so by “merging together,” with both EvBots moving forward side-by-side. When they stopped and turned around, they were so close together that they were unable to see one another’s colored cylindrical shells (which were below their cameras), causing them to move out of the corner side-by-side until their paths separated.

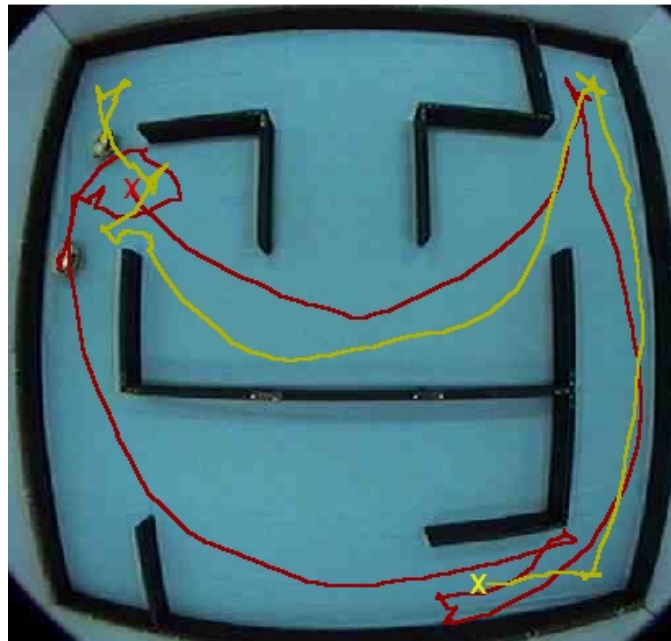


Figure 5.8 Two red EvBots (top left) clustered together and facing each other after taking the indicated paths through the maze in search of one another

The second EvBot communication experiment was more quantitative. The EvBots were configured to move in discrete steps, providing a convenient measure of the effort required for two EvBots to find each other. Two EvBots of the same color were positioned in the maze so that the first EvBot (marked with the darker, red line) would cross in front of the second EvBot (marked with the lighter, yellow line) without the first EvBot actually seeing the second. Of course, the second EvBot would see the first, but unless it could communicate, all the second EvBot could do would be to chase the first EvBot around the corner as shown in Figure 5.9. Without communication, 35 steps were required for the EvBots to cluster and face each other. With communication, however, the second EvBot could get the first EvBot's attention, causing it to stop moving forward and start turning around as shown in Figure 5.10. Communication enabled the EvBots to find each other and cluster in only 15 steps of movement, a 57% reduction in effort.

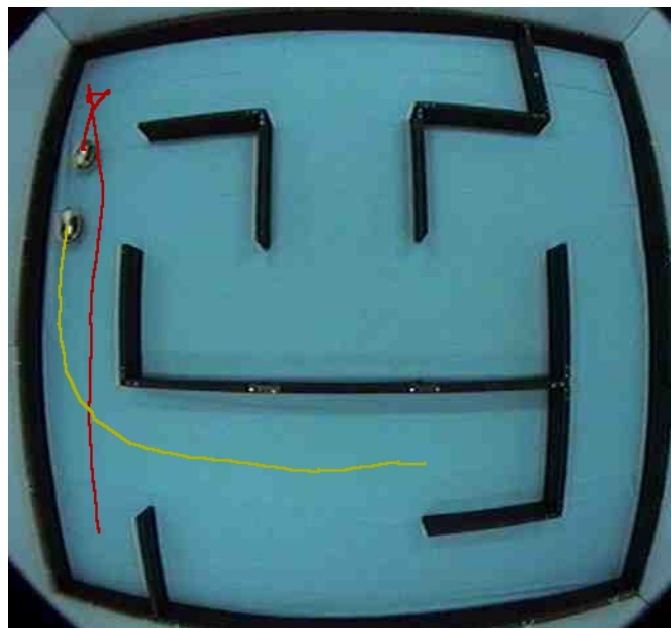


Figure 5.9 EvBots that do not communicate must chase each other until an obstruction makes the lead robot turn to face the chasing robot, causing them both to cluster together.

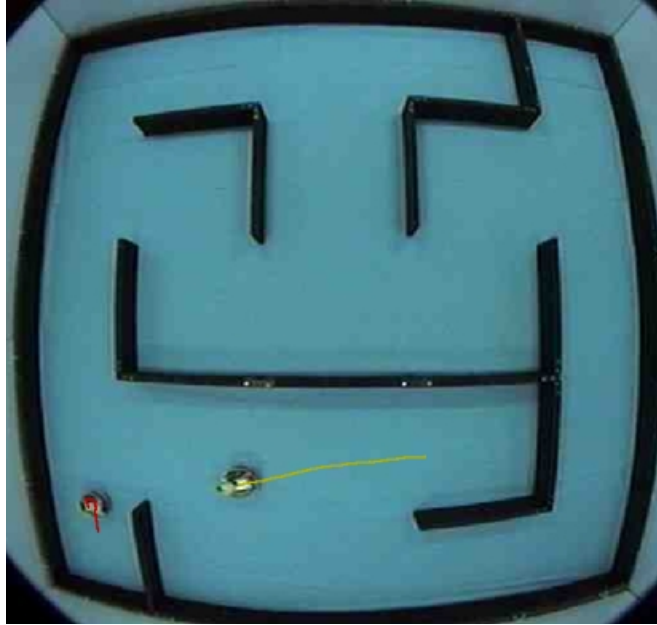


Figure 5.10 EvBots that do communicate cease to move away from each other as soon as one EvBot spots the other, causing the “lead” EvBot to gradually turn around until it finds the “chasing” EvBot.

Section 5.5 Comparison to Similar Platforms

To further analyze the capabilities of the EvBot, it is helpful to compare it to similar platforms. There are two other PC/104 based robotic-research platforms currently used in the literature: the Rascal and the Tank. Both of these were briefly discussed in Chapter 2 and are now analyzed in more detail.

Section 5.5.1 More Autonomous than the Rascal

The Rascal is a PC/104 based mobile robot approximately one foot in diameter. It has been enhanced with wireless Ethernet communications ability by Winfield and Holland, two of the most published researchers currently using it [27]. As these two point out, wireless Ethernet is an ideal communications medium for a mobile robot, and PC/104 and flash disk technologies can enable even a small robot to carry the power of

an entire desktop computer. Like their version of the Rascal, the EvBot uses all three of these key technologies. The EvBot also mimics their use of Linux as a robot operating system, allowing rapid software development through the use of standard programming tools and readily available TCP/IP network communications.

The EvBot distinguishes itself from the Rascal by being designed for fully autonomous operation, whereas the Rascal is typically remote controlled by a desktop PC. To enable advanced autonomous operation, the EvBot is equipped with a more powerful processor and an order of magnitude more RAM than the Rascal, helping the EvBot to run high-level applications like MATLAB on board. Unlike the Rascal, the EvBot does not allocate a large part of its nonvolatile storage space to a Linux distribution. Significant effort went into minimizing the size of the Infinite Atom distribution so that the EvBot would have sufficient room to store MATLAB along with several toolboxes and the user's MATLAB controllers.

Section 5.5.2 Smaller than the Tank

Carnegie Mellon's Tank is also based on PC/104 technology, but it does not capitalize on the small size of PC/104 to the extent of the Rascal or the EvBot. The Tank is roughly two feet long because it was built by retrofitting one of Tamiya corporation's remote controlled full 1/16 scale tank replicas. While using a commercially available base to provide robot mobility doubtlessly sped development of the Tank, as it did for the EvBot, the Tank is just too large to populate a colony inside a typically small lab space. Like the Rascal, the Tank is also equipped with only a 486 processor, but like the EvBot,

it utilizes a secondary MCU for low-level device interfacing, presumably freeing the main processor from simple, time-critical low-level operations.

Both the Tank and the EvBot are designed to be able to control and gather data from deployed sensing platforms. To date, work with the Tank has focused on control of the Millibot colony [28]. The EvBot is much smaller than the Tank, however, and so it would be much better suited to control of Cricket-based devices [34] and interaction with Smart Dust [33]. Another key feature shared by the Tank and the EvBot is a focus on simulation capabilities. The Tank actually carries simulation to a new level with the CyberRAVE project, allowing simulation and real hardware to interact in real-time [37]. For example, a physical Tank could navigate using data from both real and simulated sensors while fighting simulated enemies. Although the EvBot currently lacks this level of simulation functionality, the EvBot infrastructure is sufficient for its implementation.

Chapter 6. Conclusion and Future Research

Section 6.1 Concluding Remarks

The EvBot is a unique and powerful small robot designed for experiments involving evolutionary algorithms and colony behaviors. It is small and inexpensive, yet it is also robust and powerful, capable of full autonomous operation and able to provide a wealth of data. The design and construction of both the EvBot and its predecessor the JE were explained. An EvBot software package was described, and demonstrations of the EvBot's capabilities were provided.

A key feature of the EvBot is its ability to autonomously run MATLAB. To that end, a custom Linux distribution was created to fully support MATLAB while requiring a minimum of non-volatile storage space. A set of TB-interface utilities were written to enable MATLAB, as well as other programs, to control the EvBot's locomotion and to interface with its sensors and actuators. Each EvBot's local web server allows MATLAB to share data with other EvBots, while the w3c web tool allows MATLAB to both fetch data from other EvBots' web servers and to send data to a remote server equipped for data logging.

By combining the EvBot's powerful but compact hardware with its ability to locally and autonomously run controllers written entirely in MATLAB, a new class of robot has been created. Researchers with limited space and funds can now reasonably afford a colony of advanced robots. The EvBot should enable new levels of colony

experimentation involving artificial neural networks, evolvable controllers, and shared learning.

Section 6.2 Future Research

There are numerous experiments for which the EvBot would be useful. Evolved robotic colony behaviors have only begun to be explored in the literature. Experiments involving numerous autonomous robots, with each robot running genetically trained controllers, can now more easily be accomplished. The role of high-bandwidth communication in evolved controllers can be explored. There are countless experiments waiting to be performed for which the EvBot provides an adequate and useful experimental robotic platform.

The EvBot research platform could also be enhanced. The TB is currently being retrofitted with full-feedback velocity control. A new TB-interface is being written for MATLAB to alleviate MATLAB's dependence on the Linux TB-interface utilities. When finished, it should allow MATLAB to almost instantly communicate with the TB since Linux will no longer have to task swap between several processes each time a command is sent to the TB. Work is also underway to create a more advanced simulation environment for the EvBot, one that can run on a server computer and still communicate directly with a MATLAB controller running on a normal desktop. Eventually, capabilities similar to those of the Tank's CyberRAVE simulation environment may be added, potentially creating one of the most advanced robot/simulation environments available.

Chapter 7. References

- [1] Walter, W. Grey, *The Living Brain*, W. W. Norton, New York, 1963.
- [2] Rodney A. Brooks, Elephants Don't Play Chess. *Journal of Robotics and Autonomous Systems*, Vol. 6, pp. 3-15, 1990.
- [3] Rodney A. Brooks, Artificial Life and Real Robots. In F. J. Varela and P. Bourguine, editors, *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, MIT Press, Cambridge, MA, pp. 3-10, 1992.
- [4] Jean-Arcady Meyer, Evolutionary Approaches to Neural Control in Mobile Robots. 1998 IEEE International Conference on Systems, Man, and Cybernetics, Vol. 3, pp. 2418-2423, 1998.
- [5] Sean Luke, Charles Hohn, Jonathan Farris, Gary Jackson, and James Hendler, Co-evolving Soccer Softbot Team Coordination with Genetic Programming. *Proceedings of the First International Workshop on RoboCup*, Nagoya, Japan, pp. 115-118, 1997.
- [6] J. Xiao Z. Mickalewicz and L. Zhang, K. Trojanowski, Adaptive Evolutionary Planner/Navigator for Mobile Robots. *IEEE Transactions on Evolutionary Computing*, Vol. 1, No. 1, pp. 18-28, 2000.
- [7] G. Caprari, K. O. Arras, and R. Siegwart, The Autonomous Miniature Robot Alice: from Prototypes to Applications. *Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vol. 1, pp. 793-798, 2000.
- [8] Henrik Hautop Lund and Orazio Miglino, From Simulated to Real Robots. *Proceedings of IEEE International Conference on Evolutionary Computation*, pp. 362-365, 1996.
- [9] O. Michel, An Artificial Life Approach for the Synthesis of Autonomous Agents. *Proceedings of the European Conference on Artificial Evolution*, Springer-Verlag, pp. 220-231, 1995.
- [10] W. Lee, J. Hallam, and H. Lund, Applying Genetic Programming to Evolve Behavior Primitives and Arbitrators for Mobile Robots. *Proceedings of the 1997 IEEE International Conference on Evolutionary Computation*, pp. 495-499, 1997.

- [11] S. Nolfi, Adaptation as a More Powerful Tool than Decomposition and Integration: Experimental Evidences from Evolutionary Robotics. *Fuzzy Systems Proceedings. Proceedings of the 1998 IEEE Conference on Computational Intelligence*, Vol. 1, pp. 141-146, 1998.
- [12] M. Quinn, Evolving Communication Without Dedicated Communication Channels. *Proceedings of the Sixth European Conference on Advances in Artificial Life, ECAL 2001*, Prague, Czech Republic, pp. 357-367, 2001.
- [13] Akio Ishiguro, Seiji Tokura, Toshiyuki Kondo, and Yoshiki Uchikawa, Reduction of the Gap between Simulated and Real Environments in Evolutionary Robotics: A Dynamically-Rearranging Neural Network Approach. *Proceedings of the 1999 IEEE International Conference on Systems, Man, and Cybernetics*, Vol. 3, pp. 239-244, 1999.
- [14] Dario Floreano and Stefano Nolfi, Adaptive Behavior in Competing Co-Evolving Species. In Phil Husbands and Inman Harvey, editors, *Fourth European Conference on Artificial Life*, MIT Press, Cambridge, MA, pp. 378-387, 1997.
- [15] R. A. Watson, S. G. Ficici, and J. B. Pollack, Embodied Evolution: Embodying an Evolutionary Algorithm in a Population of Robots. In P. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and A. Zalzal, editors, *1999 Congress on Evolutionary Computation*, IEEE Press, pp. 335-342, 1999.
- [16] H. H. Lund, O. Miglino, L. Pagliarini, A. Billard, and A. Ijspeert, Evolutionary Robotics—A Children’s Game. *Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence*, pp. 154-158, 1998.
- [17] A. Agah and C. T. Pierik, Design and Fabrication of a Team of Robots in Hardware. *Proceedings of the 1996 IEEE IECON 22nd International Conference on Industrial Electronics, Control, and Instrumentation*, Vol. 3, pp. 1577-1582, 1996.
- [18] G.B. Parker, Generating Arachnid Robot Gaits with Cyclic Genetic Algorithms. In J.R. Koza et al, editors, *Proceedings of the Third Annual Conference on Genetic Algorithms*, Morgan Kaufmann, Madison, Wisconsin, pp. 576-583, 1998.
- [19] F. Bellas, J. A. Becerra, J. Santos, and R. J. Duro, Applying Synaptic Delays for Virtual Sensing and Actuation in Mobile Robots. *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks*, Vol. 6, pp. 144-149, 2000.
- [20] N. Jakobi, Running Across the Reality Gap: Octopod Locomotion Evolved in a Minimal Simulation. In Philip Husbands and Jean-Arcady Meyer, editors, *Evolutionary Robotics: First European Workshop, EvoRobot98*, Springer (Lecture Notes in Computer Science, Vol. 1468), pp. 39-58, 1998.

- [21] Paul E. Rybski, Sascha A. Stoeter, Maria Gini, Dean F. Hougen, and Nikolaos Papanikolopoulos, Effects of Limited Bandwidth Communications Channels on the Control of Multiple Robots. Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems, Vol. 1, pp. 369-374, 2001.
- [22] J. Spletzer, A. K. Das, R. Fierro, C. J. Taylor, V. Kumar, and J. P. Ostrowski, Cooperative Localization and Control for Multi-Robot Manipulation. Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems, Vol. 2, pp. 631-636, 2001.
- [23] Thomas Bräunl, Multi-Robot Simulation with 3D Image Generation. Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems, Vol. 2, pp. 815-820, 2001.
- [24] Thomas Bräunl and Birgit Graf, Robot Soccer with Local Vision. Pacific Rim International Conference on Artificial Intelligence, Singapore, pp. 14-23, 1998.
- [25] A. Saffiotti and K. LeBlanc, Active Perceptual Anchoring of Robot Behavior in a Dynamic Environment. Proceedings of the 2000 IEEE International Conference on Robotics and Automation, pp. 3796-3802, 2000.
- [26] G. S. Hornby, S. Takamura, J. Yokono, O. Hanagata, M. Fujita, and J. Pollack, Evolution of Controllers from a High-Level Simulator to a High DOF Robot. In J. Miller, editor, Proceedings of the Third International Conference on Evolvable Systems: From Biology to Hardware, ICES 2000, Springer (Lecture Notes in Computer Science, Vol. 1801), pp. 80-89, 2000.
- [27] A. F. T. Winfield and O. E. Holland, The Application of Wireless Local Area Network Technology to the Control of Mobile Robots. Microprocessors and Microsystems, Vol. 23, pp. 597-607, 2000.
- [28] R. Grabowski, L. E. Navarro-Serment, C. J. J. Paredis, and P. Khosla, Heterogeneous Teams of Modular Robots for Mapping and Exploration. Autonomous Robots (Special Issue on Heterogeneous Multirobot Systems), Vol. 8, No. 3, pp 293-308, 2000.
- [29] Satoshi Kagami, Mitsutaka Kabasawa, Kei Okada, Takeshi Matsuki, Yoshio Matsumoto, Atsushi Konno, Masayuki Inaba, and Hirochika Inoue, Design and Development of a Legged Robot Research Platform JROB-1. Proceedings of the 1998 IEEE International Conference on Robotics and Automation, Vol. 1, pp. 146-151, 1998.

- [30] W. L. Xu and S. K. Tso, Sensor-Based Fuzzy Reactive Navigation of a Mobile Robot Through Local Target Switching. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, Vol. 29, No. 3, pp. 451-459, 1999.
- [31] Sebastian Thrun, Wolfram Burgard, and Dieter Fox, A Real-Time Algorithm for Mobile Robot Mapping With Applications to Multi-Robot and 3D Mapping. *IEEE Conference on Robotics and Automation*, Vol. 1, pp. 321-328, 2000.
- [32] J. A. Janét, D. S. Schudel, M. W. White, A. G. England, J. C. Sutton, III, E. Grant, and W. E. Snyder, Two Mobile Robots Sharing Topographical Knowledge Generated by the Region-Feature Neural Network. *Proceedings of the 1997 IEEE International Conference on Robotics and Automation*, Vol. 3, pp. 2097-2102, 1997.
- [33] Brett Warneke, Bryan Atwood, and Kristofer S. J. Pister, Smart Dust Mote Forerunners. *14th IEEE International Conference on Micro Electro Mechanical Systems*, pp. 357-360, 2001.
- [34] Mitchel Resnick, Robbie Berg, and Michael Eisenberg, Beyond Black Boxes: Bringing Transparency and Aesthetics Back to Scientific Investigation. *Journal of the Learning Sciences*, Vol. 9, No. 1, pp. 7-30, 2000.
- [35] Alan FT Winfield, Distributed Sensing and Data Collection Via Broken Ad Hoc Wireless Connected Networks of Mobile Robots. In L. E. Parker, G. Bekey, and J. Barhen, editors, *Distributed Autonomous Robotic Systems*, Vol. 4, Springer-Verlag, pp. 273-282, 2000.
- [36] Yoshiro Hada and Kunikatsu Takase, Multiple Mobile Robot Navigation Using the Indoor Global Positioning System (iGPS). *Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vol. 2, pp. 1005-1010, 2001.
- [37] Kevin Dixon, John Dolan, Wesley Huang, Christiaan Paredis, and Pradeep Khosla, RAVE: A Real and Virtual Environment for Multiple Mobile Robot Systems. *Proceedings of the 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vol. 3, pp. 1360-1367, 1999.

Appendices

Chapter 8. Appendices

Section 8.1 Jumper Emulator (JE)

Section 8.1.1 Hardware

The JE's onboard electronic hardware was relatively simple. The controlling computer's hardware components, as well as the JE's mechanical hardware, were purchased off the shelf and so are not described here, except in the context of the JE's onboard electronic systems. The JE's camera system was assembled by simply plugging the camera into the video transmitter and providing direct battery power to both. The TB, on the other hand, was a little more complex. At the heart of the JE's hardware was its BS2 MCU, which was responsible for receiving commands from the JE's wireless digital receiver and sending commands to the JE's H-bridge motor drivers. The following were the BS2's most important specifications:

- PIC16C57 internal MCU
- 4,000 instructions per second (internal MCU runs at 20 MHz)
- 2 KB EEPROM for nonvolatile program and data storage (500 lines of code)
- 26 bytes general-purpose RAM
- 16 generic I/O pins (TTL RS232 capable) + 1 dedicated RS232 serial interface
- 7 mA drawn at 5 V (external or from the efficient internal voltage regulator)
- 24 pin DIP package

In Figure 8.1, the BS2 is the small PCB covered with surface-mount components on the right.

Figure 8.2 is a schematic of most of the JE's onboard hardware. The camera system is not shown, nor is the wireless digital receiver. The digital receiver simply connects to power and to one of the BS2's generic I/O pins (I/O pin 0 was used in the software listed in Section 8.1.2.1.2). The components were all physically mounted in a proto board as shown in Figure 8.1.

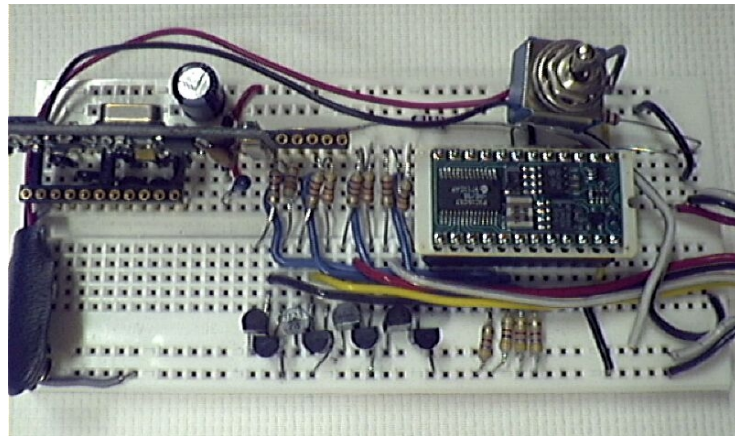


Figure 8.1 Component layout on the proto board

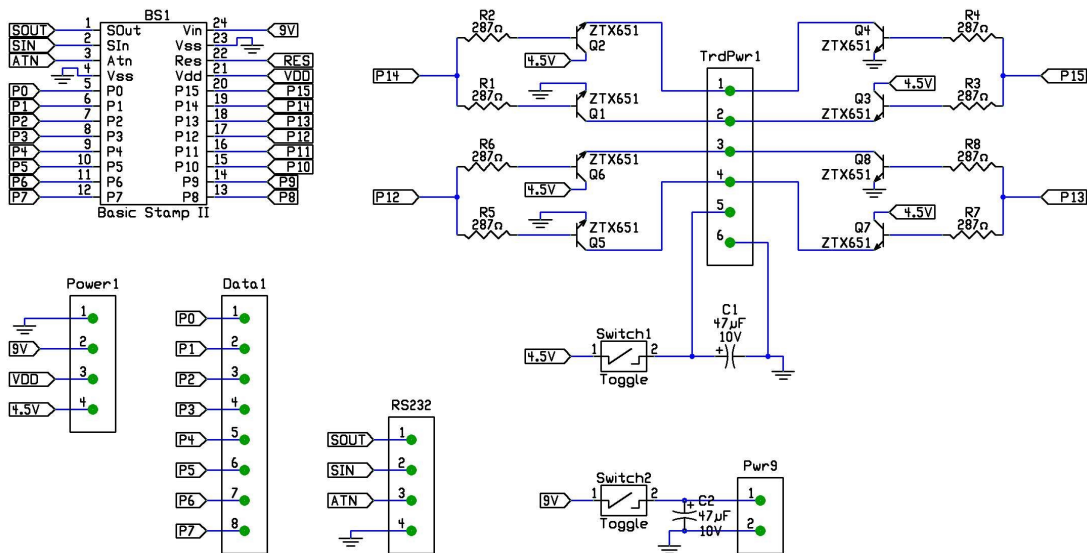


Figure 8.2 JE Schematic for the layout shown above in Figure 8.1 (The receiver attaches to P0.)

Section 8.1.2 Software

Section 8.1.2.1 JE Mobile Robot Source Code

Section 8.1.2.1.1 Description and Explanation

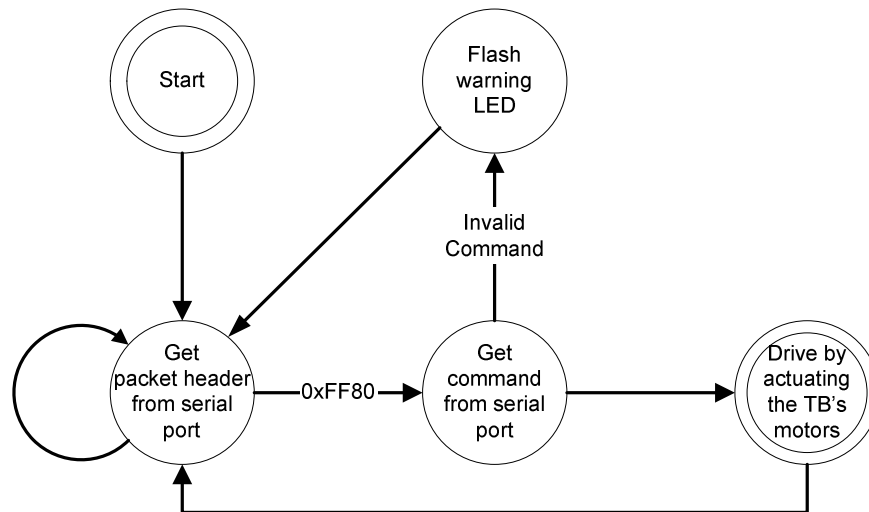


Figure 8.3 JE mobile robot source code flowchart

The BS2 code was responsible for reading commands received on the BS2's serial port (from the RS232 receiver) and actuating the TB's motors accordingly, as shown in Figure 8.3. The code continuously loops between these two primary modes of operation. The software is built around the digital wireless communication protocol that was used by the desktop computer to control the robot.

To read a command from the serial port, the PBASIC *SerIn* command is used. When called, it does not return until it has read the sequence 0xFF80, followed by one more byte. That last byte is assumed to be a command byte and is returned by *SerIn*. The command byte is compared to all known command byte values, and if there is a match, the program jumps to the command's corresponding PBASIC instructions.

Otherwise, a warning LED on the JE would be flashed before calling *SerIn* again to resume scanning the RS232 serial port input.

The PBASIC code to drive the JE is similar for all of the commands. It begins by setting the BS2's H-bridge motor-control pins such that the motors spin in the desired directions. The code then waits for a fixed duration of time before resetting all the H-bridge control pins to zero, stopping both of the motors. Control is then returned to the top of the main loop where *SerIn* looks for the next packet.

The digital wireless communication protocol, first mentioned in Section 3.2.1, is very simple. There is only one software layer in the protocol and only one type of packet. Each transmitted packet begins with the fixed header 0xFFFFF80, immediately followed by a single byte command, which terminates the packet. The first two bytes of the header are all binary ones, except for RS232's stop bit which separates the two bytes. The two bytes are used to resynchronize the RS232 receiver's bit-slicer, which was used to pull digital data from a transmitted signal. The last two bytes mark the start of a packet, and were chosen to minimize the possibility of their random occurrence, as determined by the engineers at Linx Technologies. Actually, 0x00 would have been preferable to 0x80 in the header, but the BS2 would not let its controller receive 0x00 from the serial port because the BS2 would interpret it as the end of a string and therefore not pass it. Therefore, the start of packet data is marked with 9 high bits followed by 7 low bits, instead of an even 8 bit / 8bit split, making the two-byte sequence 0xFF80. Four commands are implemented in the protocol. The command byte can be one of the values shown in Table 8.1.

Table 8.1 Commands Supported by the JE Wireless Control Protocol

Command Byte (Character Representation)	Command Description
F	Move forward for 1.5 seconds
B	Move backward for 1.5 seconds
L	Spin left for 0.4 seconds
R	Spin right for 0.4 seconds

Section 8.1.2.1.2 BS2 Source Code

```

' -----
' "Jumping" robot demo1 remote control code
' For treaded robot control
' -----
' In0          RF "RS232" input
' Out1         Bad Command LED
' Out12&13    Left tread forward&back
' Out14&15    Right tread forward&back
' -----

input 0
output 1
output 12
output 13
output 14
output 15

command VAR      byte

Rx      CON      0          ' Use P0 for RF serial link
Baud    CON      32         ' 19200 N81 Noninverted
S_TOut  CON      10000     ' wait for 10 s to receive a command

' -----

out1=0
out12=0
out13=0
out14=0
out15=0

top:
  SerIn Rx,Baud,S_TOut,S_TimeOut,[ Wait(255,128), command ]
  ' debug "Command is ", command, ", ",DEC command, cr

  if command = "F" then forward
  if command = "B" then back
  if command = "L" then left
  if command = "R" then right
  debug "Bad command: ", command, cr
  out1=1
  pause 500      ' Flash the warning LED
  out1=0
  goto top

forward:
debug "Moving forward", cr
out13=0
out15=0
out12=1
out14=1
pause 750

```

```
out12=0
out14=0
goto top

back:
debug "Moving back", cr
out12=0
out14=0
out13=1
out15=1
pause 750
out13=0
out15=0
goto top

left:
debug "Turning left", cr
out12=0
out15=0
out13=1          ' Left back
out14=1          ' Right forward
pause 333
out13=0
out14=0
goto top

right:
debug "Turning right", cr
out13=0
out14=0
out12=1          ' Left forward
out15=1          ' Right back
pause 333
out12=0
out15=0
goto top

S_TimeOut:
debug "No command in last 10 seconds.",cr
goto top
```

Section 8.1.2.2 Controlling-Computer Source Code

Section 8.1.2.2.1 Description and Explanation

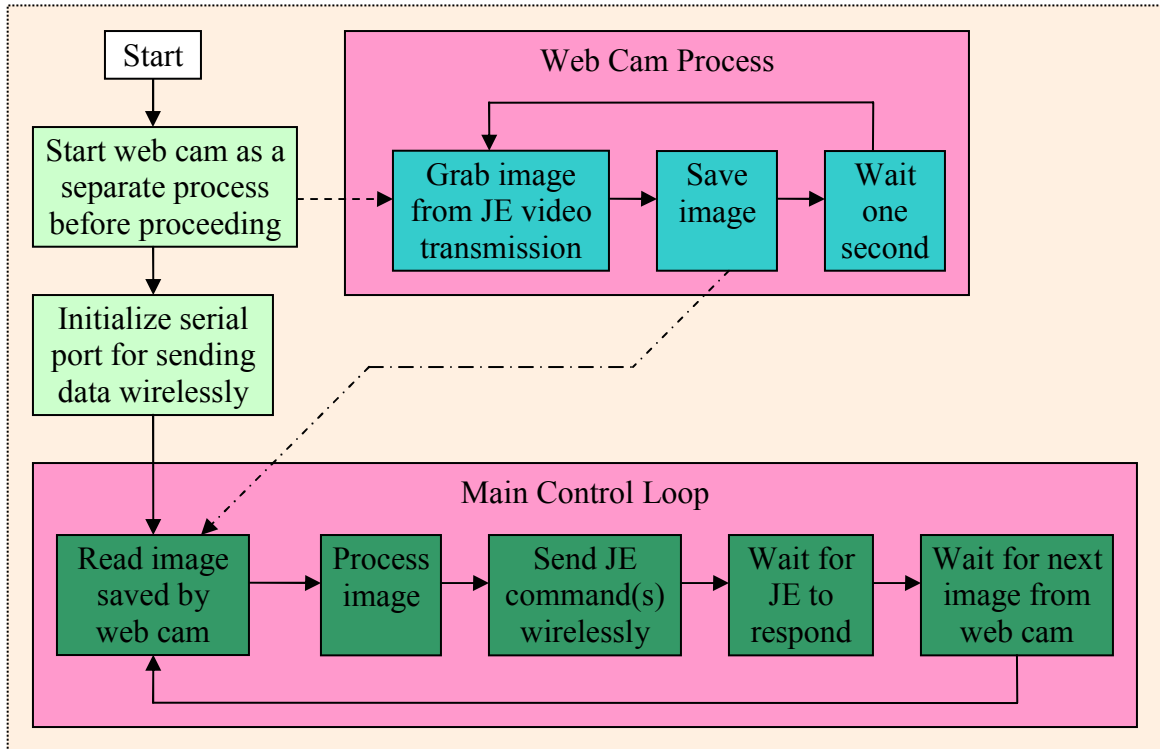


Figure 8.4 JE controlling-computer flow-chart

MATLAB, which runs on a desktop computer, completely controls the JE's behavior. Figure 8.4 shows a flowchart for the controlling computer's MATLAB behavior. MATLAB begins by starting a standard web cam application. The web cam application runs at the same time as MATLAB, and once a second it grabs the current image from the computer's video capture card and saves it to a fixed filename. After initializing the computer's serial port, MATLAB enters its main loop where it reads the last image saved by the web cam and proceeds to process it.

MATLAB's image processing toolbox is used extensively to process the images. Several image measurements and interpretations are shown on the screen to help the user adjust the controller (as shown in Figure 8.5), but edge detection is all that is used for the control of the robot. Edge detection looks for neighboring pixels in an image with strong differences in intensity, indicative of an actual edge between two different surfaces depicted in the image. Typically, edge detection also uses some algorithm(s) to prevent noise from causing small false edges and to prevent blur from masking large edges. The specific edge detector used is the Laplacian of Gaussian edge detector included in MATLAB's image processing toolbox.

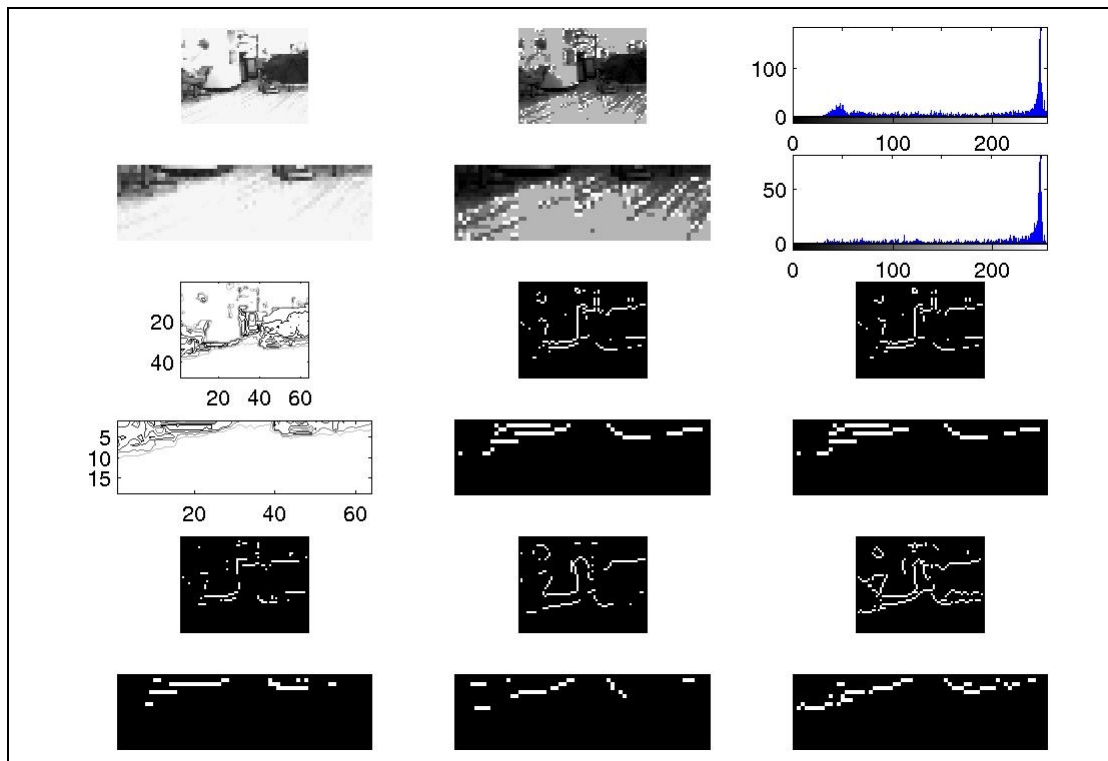


Figure 8.5 MATLAB's analysis of the image presented in Figure 3.12 showing the user (all at once) the results of several operations applied to both the entire input image and just the bottom 2/5 of it, which corresponds to the floor in front of the JE

Once the edges have been found in the image, the bottom 2/5 of the image is partitioned into three adjacent regions corresponding to the floor areas immediately in front of the robot and to its left and right. These regions would normally contain any obstacles that might interfere with the JE's next course of movement. Such obstacles can usually be identified by the edges between themselves and the floor or walls around them. By counting the number of pixels marked as edges in each region, MATLAB could determine whether or not each region contained an obstacle and therefore decide if the JE should drive straight forward, turn and drive forward, or if it must back up before turning to face another (hopefully clear) direction. This simple process could make the JE wander around a room without getting itself trapped, while theoretically also being able to simultaneously scan for surveillance targets, using vision and/or other sensors.

DOS batch file scripts were used to interface MATLAB with the computer's serial port. The scripts take a single character command as their argument, and then prefix it with a packet header before transmitting it over the serial port. A wireless RS232 transmitter was connected to the serial port and relayed the entire packets to the JE's RS232 receiver and thus to its BS2 MCU.

Section 8.1.2.2.2 MATLAB Source Code

```
disp 'Loading Web-Cam Software (WebCam32)'  
!webcam32 &  
pause(33)  
cd c:\matlab  
home;  
% This script calls two DOS batch files: com_init.bat & com_send.bat  
% They are invoked with the "!" operator and handle IO to COM1  
  
cd C:\matlab  
!com_init  
  
while 1,  
  
pause(.5);  
t=imread('WebCam.jpg');  
g=imresize(rgb2gray(t),.2); % g has dimensions 64x48
```

```

g2=imcrop(g,[0 30 64 19]); % X0, Y0, Width, Height from Top Left Corner
% g2 has dimensions 64x19, and is the bottom 2/5 of g
figure(1); imshow(t);

figure(2);
subplot(6,3,1); imshow(g);
subplot(6,3,2); imshow(histeq(g));
subplot(6,3,3); imhist(g);
subplot(6,3,4); imshow(g2);
subplot(6,3,5); imshow(histeq(g2));
subplot(6,3,6); imhist(g2);

subplot(6,3,7); imcontour(g);
subplot(6,3,8); edge(g,'sobel');
subplot(6,3,9); edge(g,'prewitt');
subplot(6,3,10); imcontour(g2);
subplot(6,3,11); edge(g2,'sobel');
subplot(6,3,12); edge(g2,'prewitt');

subplot(6,3,13); edge(g,'roberts');
subplot(6,3,14); edge(g,'log');
subplot(6,3,15); edge(g,'canny');
subplot(6,3,16); edge(g2,'roberts');
subplot(6,3,17); edge(g2,'log');
subplot(6,3,18); edge(g2,'canny');

pause(.5);
e = edge(g2,'log'); % LOG edge detect on
lower 2/5 (only want lower 7 pixels, but need a margin for edge detection to work correctly.
l_count = sum(sum(e(13:19,1:24))); % Count pixels in the Left, Forward(middle),
f_count = sum(sum(e(13:19,25:40))); % and Right regins of e. The regins do not
r_count = sum(sum(e(13:19,41:64))); % go to the top of e because e has "head room."
home;

% clear the screen for output.
disp(sprintf('Left Pixel Count: %d\nFoward Pixel Count: %d\nRight Pixel Count:
%d\n',l_count,f_count,r_count));

EDGE_PIX_LIMIT = 5;
if f_count < EDGE_PIX_LIMIT
    disp('Moving Forward...')
    !com_send F
elseif l_count < EDGE_PIX_LIMIT
    disp('Turning Left...')
    !com_send L
elseif r_count < EDGE_PIX_LIMIT
    disp('Turning Right...')
    !com_send R
else
    disp('Backing Up and Turning Left...')
    !com_send B
    pause(2);
    !com_send L
end

end
end

```

Section 8.1.2.2.3 MATLAB-Support Source Code

com_init.bat

```

@Echo off
echo Setting COM1 to 19200, N81, no flow control.
mode com1 baud=19200 parity=n data=8 stop=1 > NUL

```

com_send.bat

```

@Echo off
set cmdline=**
echo Sending "%cmdline:~1%" to COM1.
echo %cmdline:~1%>com1

rem These don't seem necessary now...
rem set cmd=%cmdline:~1%
rem echo %cmd% > com1

```


Section 8.2 EvBot Construction

The EvBot was designed as three hardware subsystems—the TB, the UB, and the PC/104 stack—that are stacked as layers on top of each other. After assembling the TB and the UB, the two should be attached before proceeding to build the PC/104 stack on top of the UB.

Section 8.2.1 Treaded Base (TB)

Section 8.2.1.1 Construction

The TB is built upon the skid-steered, tread-drive base of a toy vehicle known as the Rokenbok System. The base, as shown in Figure 8.6, must be removed from a Rokenbok toy to be used for TB construction. Three notches must be filed into the base, as shown in the figure. A custom PCB was developed and installed in the empty area between the motors at the top of the tread-drive base, as can be seen in Figure 8.7.

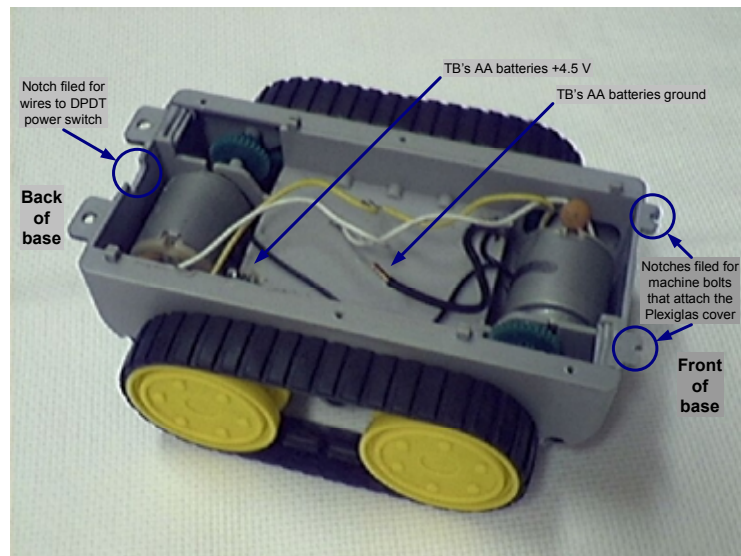


Figure 8.6 The original tread-drive base from a Rokenbok toy

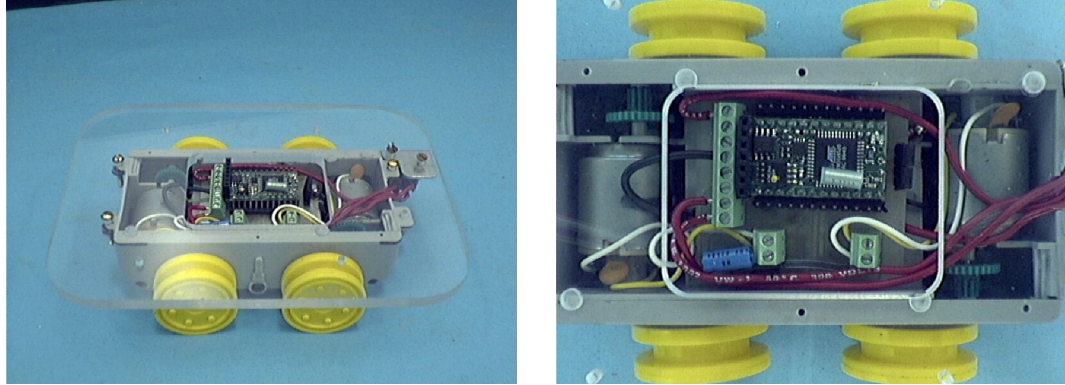


Figure 8.7 The EvBot's Treaded Base as viewed from the side and from the top

The custom PCB (Figure 8.8) contains all of the TB's electronics and electrical connections. It is made of single-sided copper and contains primarily the BasicX MCU (mounted in a socket) and two H-bridges that are made from four MSOP-8 surface-mount inverters (Zetex ZXMD63C02X), each of which consists of two high-current CMOS transistors. The PCB also contains a single 47 μ F capacitor (motor power filter), two 12-pin 0.1" SIP headers (BasicX connection pins), one 4-pin 0.1" SIP header (BasicX PWM connection pins), and three 3.5 mm pitch terminal blocks (two two-terminal blocks and one eight-terminal block). All of which are identified in Figure 8.8. The CIRCAD data files for the PCB layout are included on the CD-ROM in the "EvBot Hardware\TreadedBase\PCB\CIRCAD original" directory. An actual-size PCB layout with connection labels for the power-routing terminal-block is shown in Figure 8.9. The two two-terminal blocks are for motor connections (one motor connects to each terminal block) while the eight-terminal block is responsible for routing power as indicated by the labels. Each inverter ties both of its transistors' outputs together and connects them to its terminal of its two-terminal block for connection to one of the motor's two leads.

Warning: Do not mount the PWM connection jumpers over the PWM connection pins unless the BasicX has first been programmed to always keep pins 5 and 8 set to inputs (without pull-up resistors).

Note: Two additional pins must be soldered to the BasicX at pin-slots 26 and 27 in order to use the BasicX's PWM outputs to control the speed of TB's motors. The DIP socket used to hold the BasicX must also be modified to accept these two pins (typically done by splicing in part of another DIP socket, as shown below).

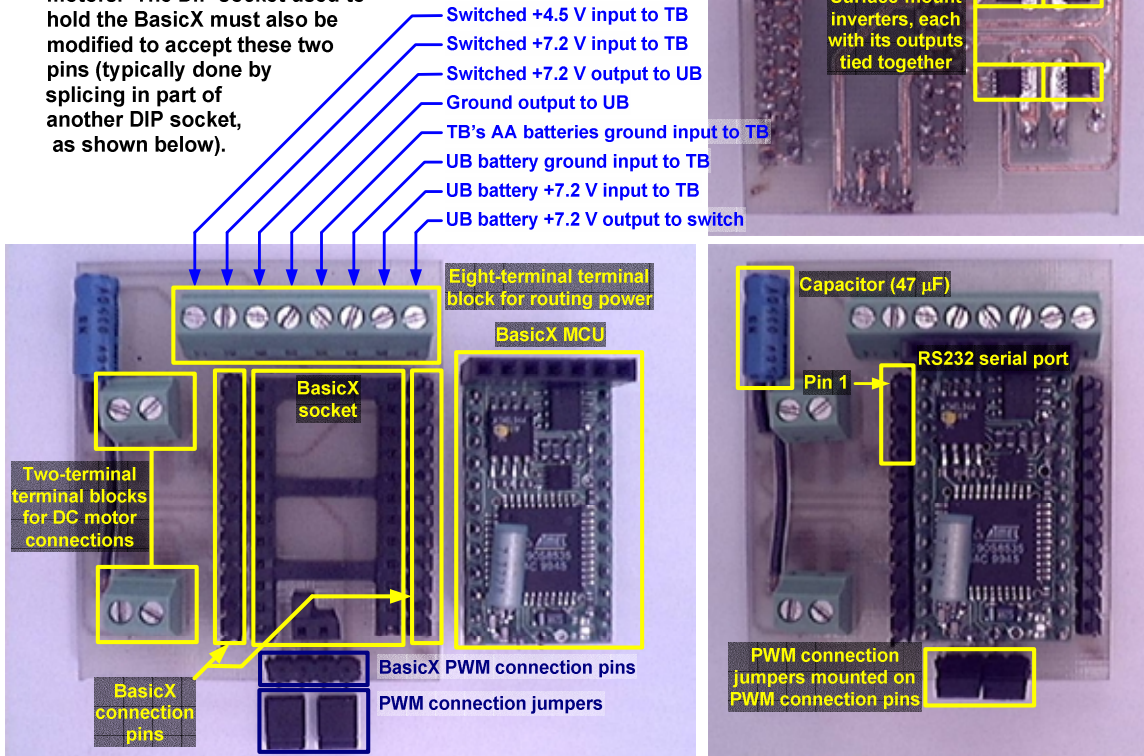


Figure 8.8 Custom PCB with labeled components as seen from the top and bottom, with and without the BasicX socketed

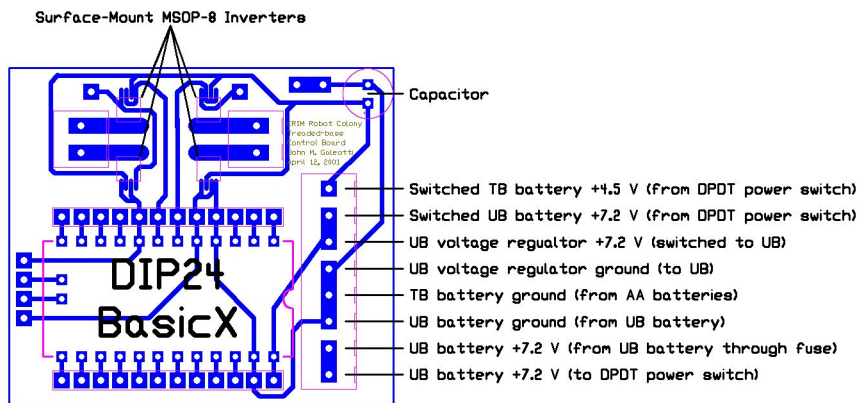


Figure 8.9 Custom PCB layout (bottom copper viewed from top) with labeled power connections

If in the future one needs more interrupt-request (IRQ) lines than the single one that the BasicX provides, hardware could be built to share the BasicX's single IRQ input among multiple IRQ outputs. Figure 8.10 shows how two encoders, each needing its own IRQ line, could be connected to the BasicX's single IRQ input. The hardware could be easily expanded to handle more than two IRQ inputs by duplicating the flip-flop and XOR hardware shown for the existing two inputs.

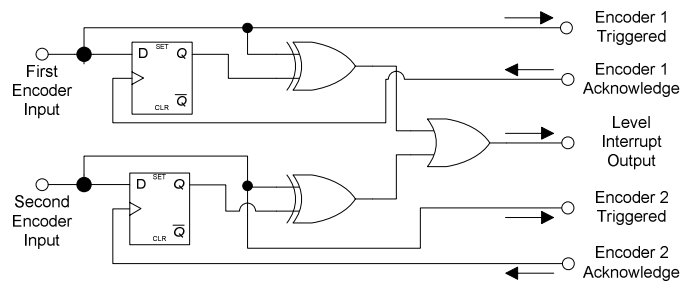


Figure 8.10 Schematic for optionally sharing the BasicX's single IRQ between two encoders outputs

The tread-drive base is covered by a piece of Plexiglas (Figure 8.11) that acts both as a bumper and as a platform for mounting the UB and the PC/104 stack. The Plexiglas has holes drilled in it at the appropriate places for mounting PC/104-spaced support standoffs. It has a hole cut in its center to allow access to the TB's PCB, and it is attached to the tread-drive base using small machine bolts. For a power switch, a double-pole, double-throw (DPDT) center-off toggle switch is mounted to the Plexiglas at the back of the TB using a special metal bracket diagrammed in Figure 8.12. Figure 8.13 shows how to physically mount the switch to the Plexiglas using the metal bracket. It also shows how to wire the switch. The two unconnected switch terminals may be used in the future for battery-charging via an external power connection.

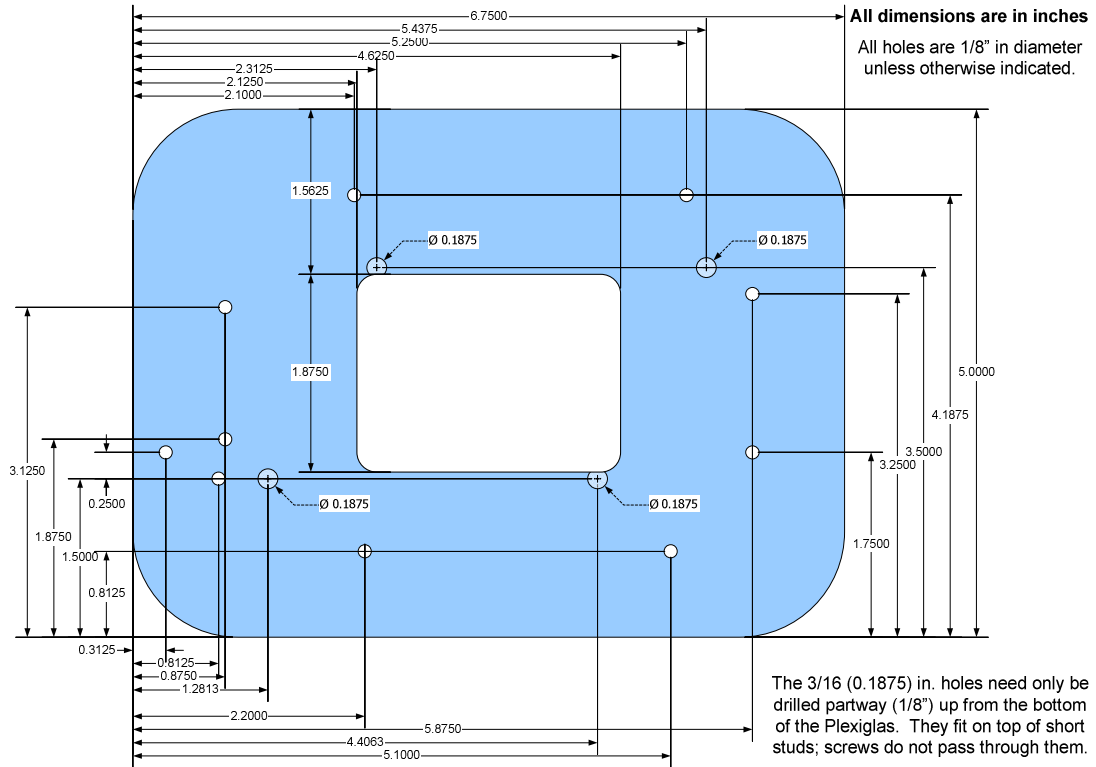


Figure 8.11 Treaded Base Plexiglas cover

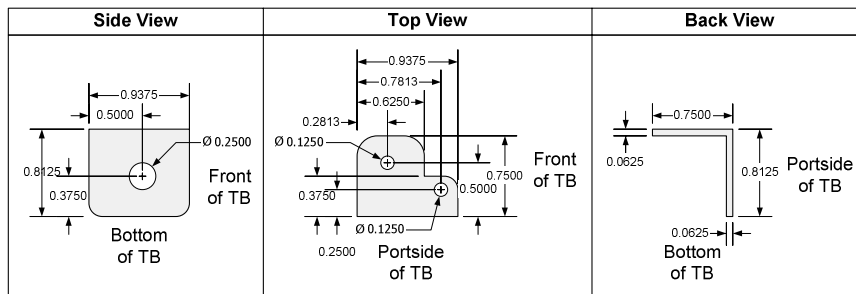


Figure 8.12 Treaded Base power switch mounting bracket (all dimensions are in inches)

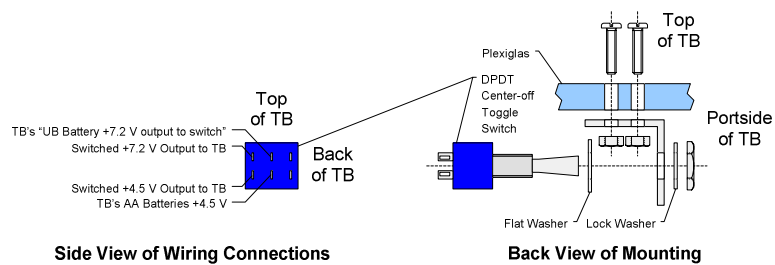


Figure 8.13 Treaded Base power switch mounting and wiring connections

Section 8.2.1.2 Test Procedure

Turn the TB's power switch off. Install AA batteries into the bottom of the TB, and connect a power supply (providing between six and twelve volts) to the TB's 7.2 V battery input (the two terminals labeled as "UB battery...input to TB" in Figure 8.8).

Attach a special EvBot TB-interface serial cable (a four-conductor serial cable designed to connect to the four RS232 pins of the BasicX at one end and to a computer at the other) to a computer running Linux and to the TB. Pin one of the serial cable's four pin connector (which should be designated by a white dot) goes to pin one of the BasicX connection pins, as identified in Figure 8.8. Place the TB on top of an object so that all of the wheels are free to turn without them moving the TB. Turn on the base and then login on the computer running Linux and run the `tread_test` program located in the "Support Software\EvBot development station" directory of the CD-ROM. The program is stored in the `TreadedBaseTest.tgz` archive, which should be copied to an installation directory on the computer and then extracted by entering "`tar xzvf TreadedBaseTest.tgz`" at the command prompt. To run the program, enter the newly created `TreadedBaseTest` directory and then enter `./tread_test` at the command prompt. The wheels should run at a "high" speed forward, backwards, right, left and then stop. They should then go through the same sequence at a much slower speed. If the test fails, use an oscilloscope to check for "correct" PWM waveforms on BasicX connection pins 5 through 8 as shown in Figure 8.14. If all four outputs are always correct, there is a problem with either the motors' wire connections or the surface-mount transistors. Otherwise, there is either a problem with the RS232 serial interface or there is a software problem.

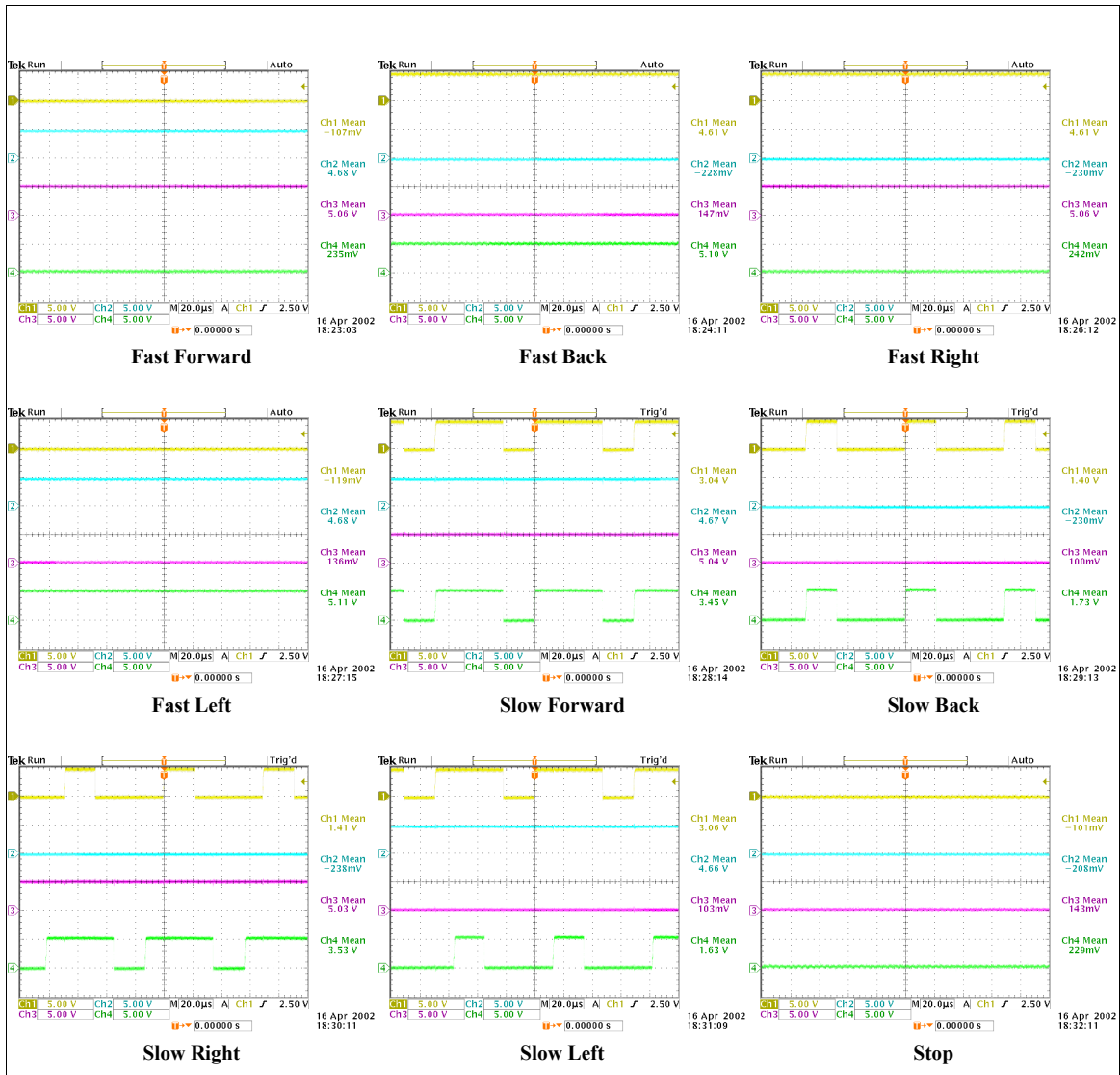


Figure 8.14 PWM waveforms on BasicX connection pins 5 through 8 of the TB (connected to channels 1 through 4 on the oscilloscope)

Section 8.2.2 Utility Board (UB)

Section 8.2.2.1 Construction

The UB is a large PCB mounted on top of the TB for physical and electronic support of the PC/104 stack. It forms the base of the PC/104 stack, itself being attached to the TB using PC/104-spaced standoffs. It also attaches to the TB at the back of the

EvBot using an additional female/female standoff at the location of one of the TB's Plexiglas-support machine bolts for added rigidity. The PCB layout is shown in Figure 8.15, and CIRCAD data files for the layout are included on the CD-ROM in the "EvBot Hardware\UtilityBoard\PCB\CIRCAD original" directory.

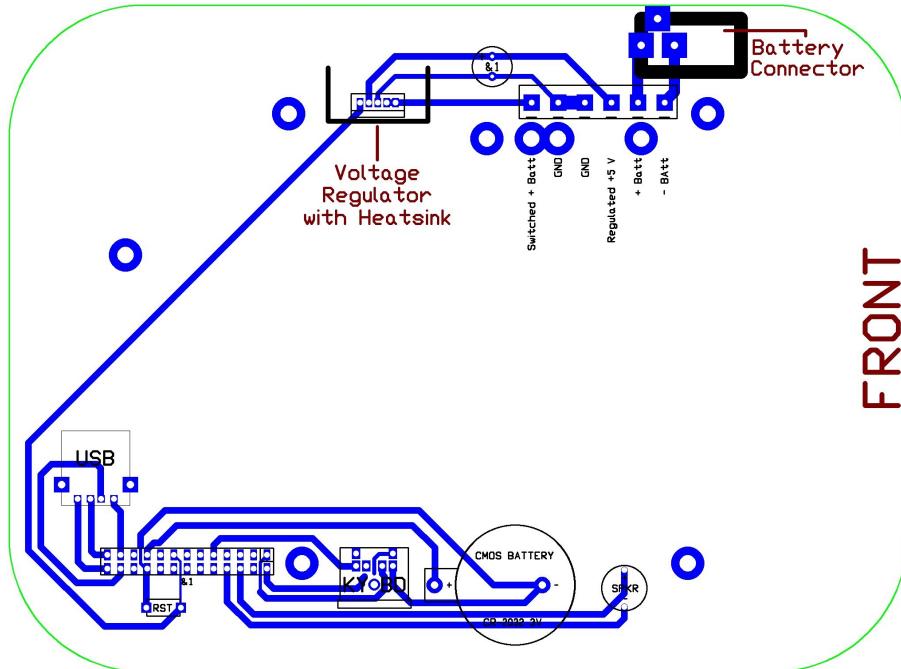


Figure 8.15 Utility Board PCB layout (bottom copper viewed from top) with labeled components

The most important component of the UB is its power supply. The UB uses Micrel's 5 V, 1.5 A LDO voltage regulator model MIC29151-5.0BT. This regulator also has a built in voltage monitor that can signal an external device if the output voltage goes out of range, such as when the supply battery is discharged. On the UB, the voltage-monitor signal is wired to the MZ104 motherboard's reset input, insuring that the PC/104 stack will not try to write to flash storage when insufficient voltage is available, as this could permanently damage the flash electronics. The regulator is powered by a 7.2 V,

3000 mAh Ni-MH battery that connects to the UB's battery connector using a 2.1 mm DC power plug. A 7.2 V battery is used because the PC/104 stack can occasionally draw extra-high current surges, and a 6V battery does not provide the regulator with adequate headroom to handle the surges, causing the PC/104 stack to reset. Because the battery is the heaviest item mounted on the TB, it is mounted low (directly on top of the UB's PCB), and it is centered between the PC/104-support standoffs to maintain a stable center of gravity. A fused link connects the battery's terminal block output (labeled "+ Batt" on the UB) to the TB's power-switch input (labeled "UB battery +7.2 V input to TB" in Figure 8.8).

The UB is also responsible for connecting several components to the MZ104 motherboard via the motherboard's utility port. The UB carries a 3 V Li battery that continuously supplies the MZ104's real time clock with power and preserves CMOS settings when the EvBot is turned off. The UB also carries a small PC-speaker and connectors for a keyboard and a USB device. (If that USB device is a hub, then many USB devices can be connected through it to the UB.)

Section 8.2.2.2 Test Procedure

Connect an adjustable voltage supply to the switched 7.2 V battery input of the UB (pins one and two of the terminal block, labeled "Switched + Batt" and "GND") after setting the supply to 7.2 V. Make sure the UB's voltage regulator is supplying 5 V at the PC/104 power connections (pins three and four of the terminal block, labeled "GND" and "Regulated +5 V"). Make sure that the UB's regulator is not pulling the reset pin all the way down to ground by measuring the voltage at the front pin (which is accessible from

the top of the UB) of the manual reset switch (which is at the bottom left in Figure 8.15). The voltage should be floating and not all the way down to ground. While measuring the reset voltage, press and hold the reset switch; the reset voltage should now be pulled down to ground. Release the switch (the reset voltage should float again). Reduce the output of the voltage supply to 4.5 V; the reset voltage should be pulled to ground again. Raise the voltage back to 7.2 V and make sure that the reset voltage once again floats.

Test each electronic connection from the pins of the utility-port connection header (located beside the reset switch) to the components to which they connect. Check for shorts between all physically adjacent connections and traces on the UB.

Section 8.2.2.3 Connection Cables

Two cables are necessary to connect the TB and the UB to the PC/104 stack. The cables should be assembled before attaching the UB to the TB. The TB is controlled by the PC/104 stack via a special RS232 serial cable. The cable is produced by attaching two Molex connectors to a nine-inch, four-conductor ribbon cable, as shown in Figure 8.16. A four-by-one Molex connector is connected to one end, and a five-by-two Molex connector is connected to the other end; looking at the cable end of the five-by-two Molex connector, the cable connects to the top row at the left four pin sockets. White pin-one marker dots should be placed at the indicated locations on the Molex connectors.



Figure 8.16 RS232 serial cable layout

The MZ104 PC/104 motherboard's utility port connects to the UB using a 3.5 inch 26-pin ribbon cable. Standard 26-pin press-on female connectors should be connected to both ends of the cable, taking care to align pin one of each connector (usually designated by a triangle) with the red stripe on the ribbon cable.

Section 8.2.2.4 Connection to the Treaded Base

Several wires need to be run from the TB to the UB. Make the connections shown in Figure 8.17, paying attention to wire color, which distinguishes ground wires (shown in green) from the others (shown in red). All of the wires connecting the TB to the UB should each be run through its own hole located in front of the UB's terminal block. As shown in the figure, the "- Batt" wire from the UB should be routed around the other wires to the hole not in front of the terminal block; it may be easiest to attach this wire last. Be sure to use the wires already attached to the inline fuse when making its connections. The serial PC104 interface cable should be attached next. Place its four-pin female connector over the "RS232 serial port" of the BasicX (identified in Figure 8.8), with the cable's white dot aligned with "Pin 1" of the RS232 serial port.

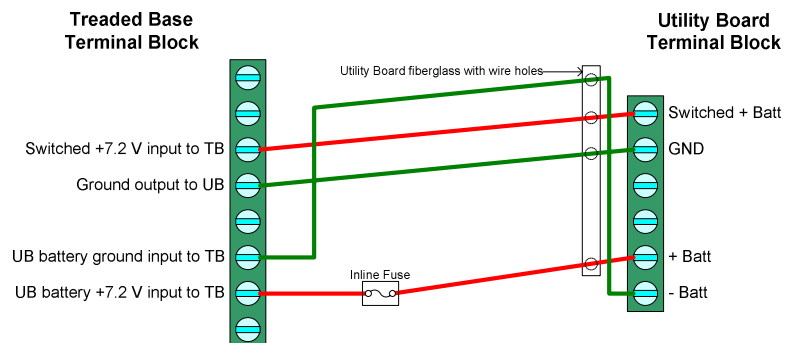


Figure 8.17 Electrical power connections between the Utility Board and the Treaded Base

Mount a pair of male/female (M/F) 5/8 inch standoffs through each of the four PC104 support holes (labeled in Figure 8.18) by inserting their screw-tip through each of the 4 UB holes from the bottom and fastening the standoffs from the top with the female end of 3/4 inch standoffs. Use a screw to attach a female/female 5/8 inch standoff to the Plexiglas cover of the TB by inserting the screw up through the Plexiglas hole located 3.125 inches from the bottom on the left side in Figure 8.11 and fastening it on top of the Plexiglas with the female/female standoff. Next, prepare to attach the UB to the base by twisting the wires from the UB to the base into a loop and laying the inline fuse partway into the recessed cavity of the TB, as shown in Figure 8.19. Making sure that the fuse stays in place and that all wires except the serial cable stay between the UB's standoffs, line-up the UB's "hole for female/female standoff" (also identified in Figure 8.18) with the female/female standoff on the TB and fasten them together using another screw; two more screws should be used to fasten the front two UB standoffs to the Plexiglas through the two PC104 holes at the front of the Plexiglas. The serial cable should protrude from the front of the robot. Insert a CR2032 3V coin battery into the UB's battery holder with the + side facing up. Finally, connect a 4 inch red wire to the UB's "Regulated +5 V" terminal and a 4 inch green wire to the UB's unused "GND" terminal.

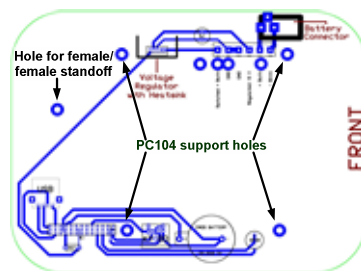


Figure 8.18 Utility Board support hole locations

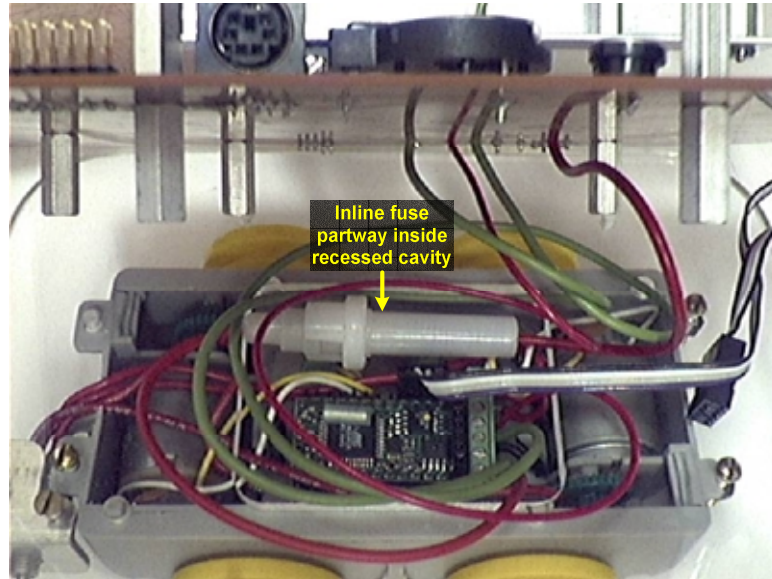


Figure 8.19 Wire placement for connecting the Utility Board to the Treaded Base

Section 8.2.3 PC/104 Stack

Section 8.2.3.1 Component Preparation

Before assembling the PC/104 stack on top of the UB, the MZ104 PC/104 motherboard needs to be prepared, and support structures for the optional USB camera need to be built. Care should be taken when handling the electronic components of the PC/104 stack to protect them from electrostatic discharge. To prevent static damage, anyone working with the electronic components should touch both of their hands to a well grounded metal object to discharge any static electricity they may have accumulated, and they should occasionally retouch such a well grounded object as they work. Optionally, they could wear an antistatic grounding strap instead.

The MZ104 PC/104 motherboard needs to be prepared for use in the EvBot. Depending on how the MZ104 was ordered, some of these steps may have already been performed, but adherence to all of these steps should still be verified to ensure correct

EvBot behavior. To begin with, the MZ104 should be configured to use a 100 MHz core CPU clock (at the time of writing, this is the factory default). A 32-pin DIP 8 MB DiskOnChip (DOC) should be installed in the MZ104's DOC socket. (A larger DOC may be used if desired, provided it is compatible with the MZ104. Consult the MZ104 documentation.) A SO-DIMM SDRAM module containing at least 64 MB of RAM must also be installed on the MZ104. The SO-DIMM should be designed for the MZ104; a standard laptop SO-DIMM will not work. Finally, it is a good idea to verify that each jumper on the MZ104 is set to the desired setting. Consult the MZ104 manual. Figure 8.20 labels the DOC and SDRAM SO-DIMM installed on a MZ104.

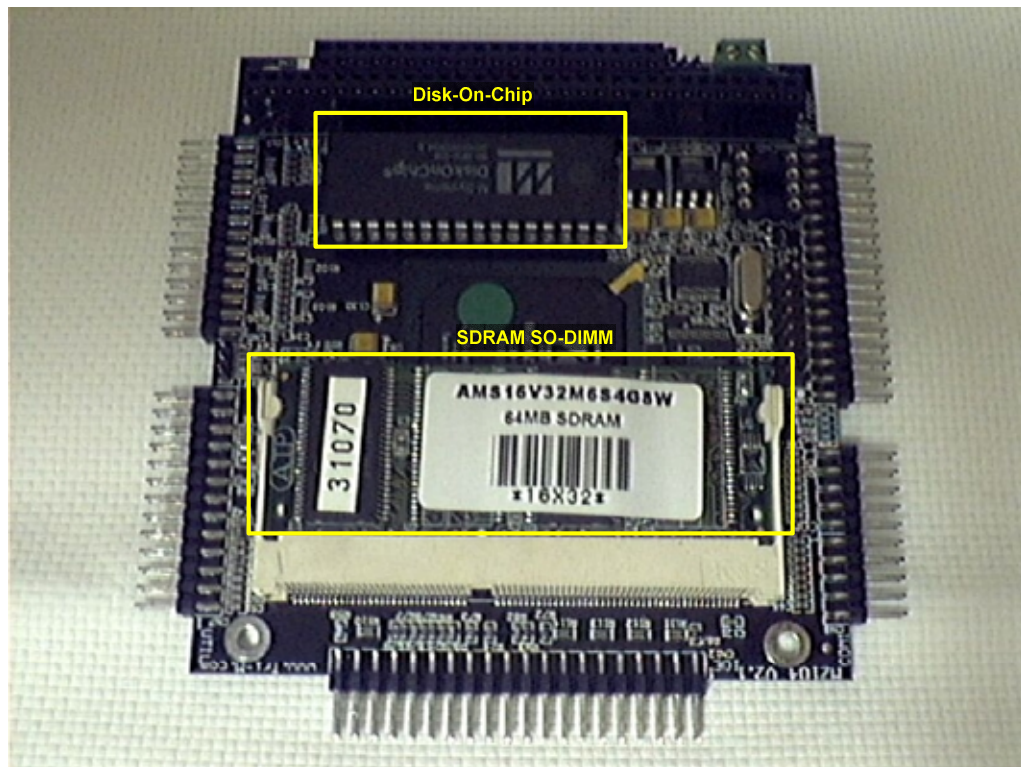


Figure 8.20 MZ104 with labeled DOC and SDRAM

The camera support structures are diagrammed in Figure 8.21. The figure also shows how to attach the camera mounting plate to the camera supports, but it will be easiest to leave it unattached until after the PC/104 stack is fully assembled.

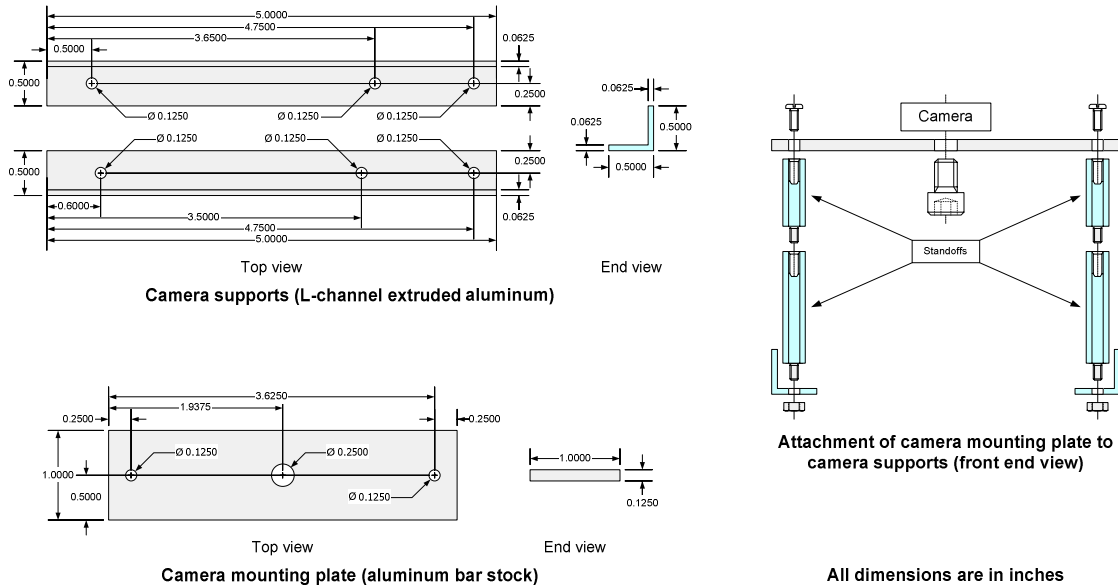


Figure 8.21 Camera support structures

Section 8.2.3.2 Stack Assembly

If these instructions have been correctly followed to this point, the EvBot should look like Figure 8.22. The 3.5 inch ribbon cable described in Section 8.2.2.3 should be connected to the UB's utility port such that its red stripe is toward the EvBot's front and the cable hangs off to the EvBot's right side. Place the camera-mount L-beams on top of the UB's standoffs. The top beam in Figure 8.21 should be on the left side and the other beam should be on the right. Secure the L-beams with 5/8 inch M/F standoffs. The EvBot should now look like Figure 8.23.

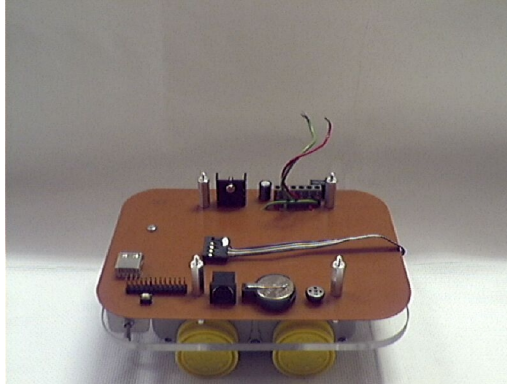


Figure 8.22 EvBot prior to assembling PC/104 stack

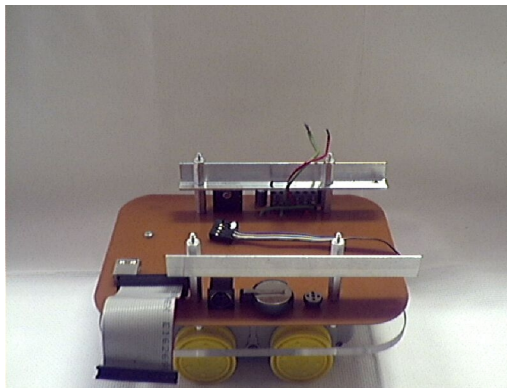


Figure 8.23 EvBot with support components for the PC/104 stack

Place the PC104 motherboard on top of the 4 uncovered standoffs such that the CPU is on top. It will only fit when rotated correctly. Secure it with four more M/F 5/8 inch standoffs. Route the PC104 stack's two power supply wires from the UB's terminal block (the two center terminals) up and out between the L-beam and the motherboard (see Figure 8.24); fasten them to the motherboard's power terminal block such that the red wire is to the back of the EvBot. Wrap the PC104 utility cable around the outside of the L-beam and loop it down to the utility connector on the motherboard such that the red stripe is on the right side as shown in Figure 8.25. Connect the free end of the PC104 serial cable to the COM1 port of the motherboard such that its white dot matches pin one,

which is toward the center of the motherboard's 2-port header. The EvBot should now look as shown in Figure 8.24 and Figure 8.25.

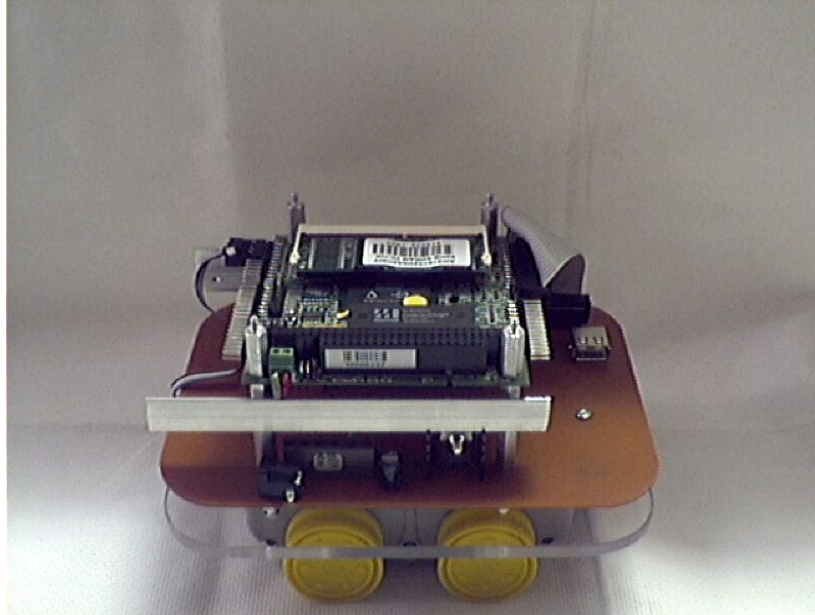


Figure 8.24 Port side of EvBot showing one view of wire connections to the MZ104 motherboard

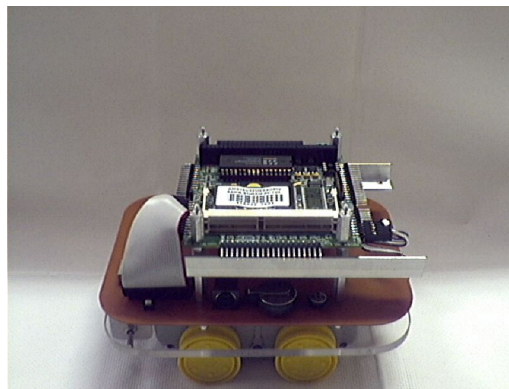


Figure 8.25 EvBot with MZ104 motherboard mounted and wires connected

Carefully place the PC104 PC-Card interface board on top of the motherboard and standoffs such that the large block of pins on its bottom is inserted into the large black receptacle on top of the motherboard; secure with four M/F 3/4 inch standoffs. Attach

the camera mounting plate to the camera supports as indicated in Figure 8.21. Insert the ATA flash card into the bottom PC-Card slot and the wireless Ethernet PC-Card into the top PC-Card slot. The EvBot should now look like the image to the right in Figure 8.26.

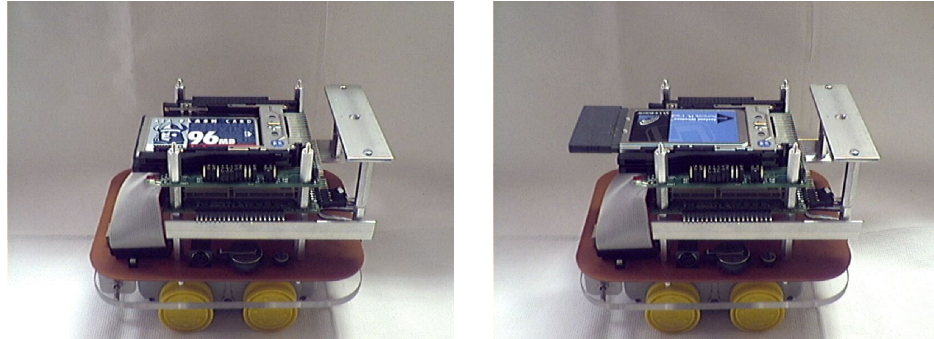


Figure 8.26 EvBot with camera mounting plate and PC-Card interface installed (before and after inserting the wireless Ethernet PC-Card)

The USB camera should not be mounted yet. Once the EvBot's software has been installed (instructions are provided in Section 8.5), the USB camera can be attached using a standard camera-mount screw through the bottom of the camera mounting plate as indicated in Figure 8.21. When wrapping the camera's USB cord around the standoffs between the TB and the UB as shown in Figure 8.27, care should be taken not to pull the cord too tightly against the wireless Ethernet card.

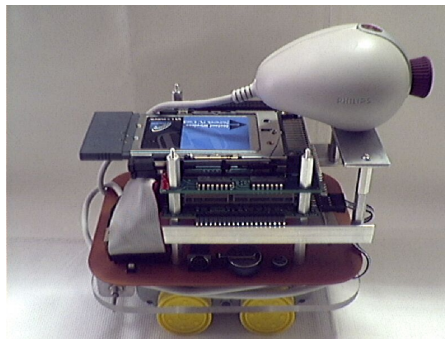


Figure 8.27 EvBot fully assembled with USB camera mounted

Section 8.3 EvBot Software

Section 8.3.1 TB Source Code

Section 8.3.1.1 Description and Explanation

Because the TB uses a BasicX as its MCU, the firmware is written in the Basic Express language (BasicX code). Basic Express is an enhanced subset of Visual Basic that allows rapid coding of support for new devices as they are added to the TB. The BasicX code is responsible for overseeing EvBot locomotion, actuating any devices attached to the TB by the user, and reading any simple sensors attached to the TB. It must interface with the PC/104 stack to carry out these operations in accordance with the instructions of the main controller, since the main controller runs on the PC/104 stack. The interface between the PC/104 stack and the TB follows the command-response paradigm of the TB interface protocol (TBIP).

A high-level flowchart of the current BasicX code is provided in Figure 8.28. At present, the BasicX code implements a few essential commands, all of which perform a short task and then return. As such, these commands are completely contained (and serialized) within the main loop (the primary task) that handles the TBIP. The main loop uses a finite state machine (FSM) to iteratively receive and process commands from the PC/104 stack. The FSM has one state for the processing of each command. Because the BasicX is capable of multitasking and interrupt processing, it is possible for one command's execution within the main loop to start, stop, or control a separate task that runs concurrently with the main loop. Such a separate task could perform some periodic function, possibly in response to hardware-triggered events with the use of interrupts, as

would be required, for example, to implement closed loop velocity control. Several helper functions are also included for convenience in the existing TB firmware.

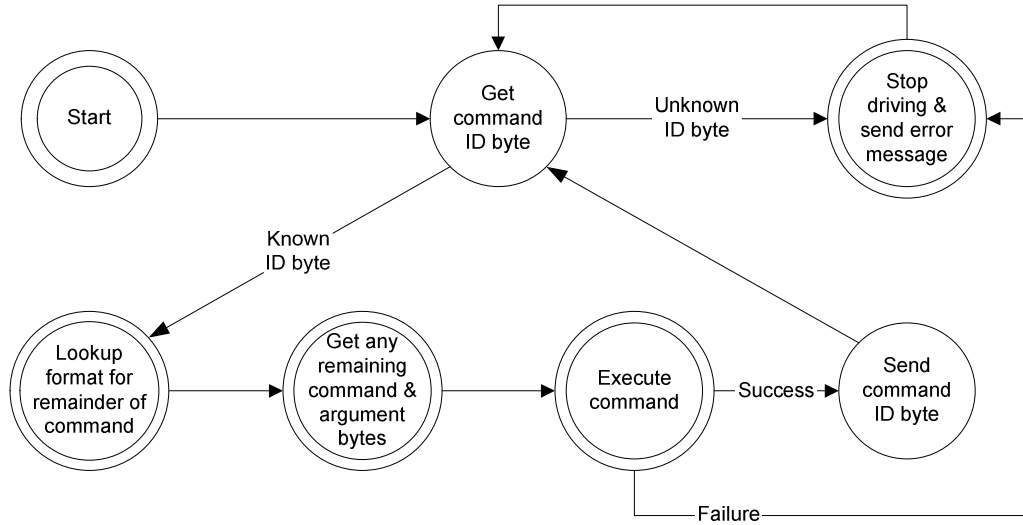


Figure 8.28 TB BasicX firmware high-level flowchart for the main loop

Section 8.3.1.2 BasicX Code

```

'-----
Option Explicit

Const InputBufferSize As Integer = 25 ' 16-byte buffer.
Const OutputBufferSize As Integer = 25 ' 16-byte buffer.

Private InputBuffer(1 To InputBufferSize) As Byte
Private OutputBuffer(1 To OutputBufferSize) As Byte

Const ASCII_LF As Byte = 10
Const ASCII_CR As Byte = 13
Const ASCII_plus As Byte = 43
Const ASCII_minus As Byte = 45
Const ASCII_decimal As Byte = 46
Const ASCII_zero As Byte = 48

Const LeftFront As Byte = 7
Const LeftBack As Byte = 8 ' pin 8 = PWM pin 27 when jumpered
Const RightFront As Byte = 6
Const RightBack As Byte = 5 ' pin 5 = PWM pin 26 when jumpered

Const RightPWM As Byte = 26 ' Also green LED
Const LeftPWM As Byte = 27

Const PinOC1A As Byte = 26 ' Also green LED
Const PinOC1B As Byte = 27

Const RedLED As Byte = 25

Const LEDon As Byte = 0
Const LEDoff As Byte = 1
Const High As Byte = 1
  
```

```
Const Low As Byte = 0
```

```
-----  
' This code uses an FSM to parse the serial input data commands. The FSM's  
' state variable is CurrentCommand and its initial state searches for single  
' letter commands. When a command is found, it is either executed or, if it  
' needs arguments, the FSM changes to that command's state where the FSM reads  
' and parses serial input data until it has read all the command's input  
' parameters. The FSM then switches back to its initial state. Note that a  
' command's state may actually be another FSM, with CurrentParameter as the  
' nested FSM's state variable. On each iteration of the main loop, the master  
' FSM is "called" if serial data is waiting and any "polling" code is always  
' executed.  
'  
' Serial output should be prefixed with the letter of the command that  
' generated that output, and should be terminated with a newline. If no  
' command is responsible for generating the serial output, then some unique  
' identifying letter (one not used for any commands) should prefix the serial  
' output. All commands must generate 1 and only 1 line of output.  
'  
' As an example, consider the command to read digital input pin #3: "R3"  
' R means read a digital input pin and 3 is the "R" command's only argument.  
' If pin #3 were currently high, then "R3" would return over the serial port:  
' "R1" and it would be up to the calling program to know that this was in  
' response to the last "R" command it called, "R3", yielding the total info  
' of pin #3 and a logic value of 1.  
-----  
Sub Main()  
  Dim ByteReady As Boolean  
  Dim NewByte As Byte  
  Dim CurrentCommand As Byte ' master FSM state variable  
  Dim CurrentParameter As Byte ' argument-parsing, nested FSM state variable  
  
  ' *****  
  ' Initialization Code  
  ' *****  
  
  CurrentCommand=0 ' NULL means not currently processing any command  
  CurrentParameter=1 ' 1 means ready to read first parameter, if necessary  
  
  Call PutPin(RedLED, LEDOff)  
  Call OpenSerialPort(1, 19200)  
  Call InitializePWM(1)  
  
  ' *****  
  ' The Main Loop  
  ' *****  
  do  
    Call GetByte(NewByte, ByteReady)  
  
    ' =====  
    ' The serial-input-parsing, command-finding FSM  
    ' =====  
  
    ' -----  
    ' This is where we search for single letter commands.  
    ' -----  
    If ( ByteReady And (0=CurrentCommand) ) Then ' Start a new command  
      Select Case NewByte  
  
        Case 80,112 ' Capital P and lowercase p: Set PWM  
          ' Byte format is: [P] [(LeftDir<<1)|RightDir] [LeftSpeed] [RightSpeed]  
          CurrentCommand = 80 ' We have arguments to read  
  
        Case 82,114 ' Capital R and lowercase r: Read a digital input pin  
          ' Byte format is: [R] [I/O pin #, A,a=10, B,b=11, etc]  
          CurrentCommand = 82 ' We have arguments to read  
  
        Case 83,115 ' Capital S and lowercase s: Emergency Stop  
          Call PutPin(LeftFront, 1)  
          Register.OCR1BH = 0  
          Register.OCR1BL = 255  
          Call PutPin(RightFront, 1)  
          Register.OCR1AH = 0
```

```

        Register.OCR1AL = 255
        ' We've no arguments to parse, so CurrentCommand stays = 0

    Case Else
        Debug.Print "--ERROR--"
        Debug.Print "New Command = " ; CStr(NewByte)
        Call setPWMfloat(LeftPWM, 0.0)
        Call setPWMfloat(RightPWM, 0.0)
        ' We've no arguments to parse, so CurrentCommand stays = 0
    End Select

' -----
' This is where commands parse their arguments
' -----
ElseIf ( ByteReady ) Then ' Parse the current command's arguments
    Select Case CurrentCommand

        ' Capital P: Set PWM
        '-----
        Case 80
            ' Byte format is: [P] [(LeftDir<<1)|RightDir] [LeftSpeed] [RightSpeed]
            Dim LeftSpeed As Byte
            Dim RightSpeed As Byte
            Dim LeftDir As Byte
            Dim RightDir As Byte

            'NewByte = (NewByte OR bx10000000) ' Scale PWM for keyboard input
            Select Case CurrentParameter
                Case 1 ' left and right direction
                    LeftDir = GetBit(NewByte,1)
                    RightDir = GetBit(NewByte,0)
                    CurrentParameter=2
                Case 2 ' left speed
                    If (0 = LeftDir) Then
                        LeftSpeed = NewByte
                    Else
                        LeftSpeed = Not NewByte
                    End If
                    CurrentParameter=3
                Case 3 ' right speed
                    If (0 = RightDir) Then
                        RightSpeed = NewByte
                    Else
                        RightSpeed = Not NewByte
                    End If
                    ' Finally, we're ready to control the motors
                    Call PutPin(LeftFront, LeftDir)
                    Register.OCR1BH = 0
                    Register.OCR1BL = LeftSpeed
                    Call PutPin(RightFront, RightDir)
                    Register.OCR1AH = 0
                    Register.OCR1AL = RightSpeed

                    Call PutByte(80) ' Output "P"
                    Call NewLine()

                    ' Go back to the initial FSM state now
                    CurrentCommand=0
                    CurrentParameter=1
            End Select ' PWM CurrentParameter

        ' Capital R: Read a digital input pin
        '-----
        Case 82
            ' Byte format is: [R] [pin #, A,a=10, B,b=11, etc]
            Dim PinNum As Byte
            Dim PinValue As Byte

            PinNum = AtoI( NewByte )

            ' Make sure we don't tristate the serial port or the PWM outputs
            If ( PinNum > 8 ) Then
                'Call PutPin(PinNum, bxInputTristate) ' set pin to high-Z input
                Call PutPin(PinNum, bxInputPullup) ' set pin to 120k pull-up
            End If
            PinValue = GetPin(PinNum)

```

```

        Call PutByte(82)                ' Output "R"
        Call PutByte(PinValue+48)      ' Convert value to ASCII
        Call NewLine()                 ' Output Newline

        CurrentCommand=0

        ' Bad Input: Stop PWM
        '-----
        Case Else
            Debug.Print "--ERROR--"
            Debug.Print "Current Command = "; CStr(CurrentCommand)
            Call setPWMfloat(LeftPWM, 0.0)
            Call setPWMfloat(RightPWM, 0.0)
            CurrentCommand=0
        End Select

    End If ' ByteReady And (0=CurrentComand) ElseIf ByteReady

    ' =====
    ' Polling code, which is called on every loop.
    ' =====

    ' There currently isn't any.

Loop
End Sub
'-----

'-----
' Convert an ASCII base-36 number (0=0...9=9,A=a=10...z=Z=35) to binary.
'-----
Public Function AtoI (ByVal char As Byte) As Byte
    If ( (char>=65) AND (char<=90) ) Then ' between A and Z
        AtoI = char - 55
    ElseIf ( (char>=97) AND (char<=122) ) Then ' between a and z
        AtoI = char - 87
    ElseIf ( (char>=48) AND (char<=57) ) Then ' between 0 and 9
        AtoI = char - 48
    Else ' ERROR TRAP
        AtoI = 255
    End If
End Function
'-----

'-----
' Control the two PWM signals.
'-----
Public Sub setPWM( _
    ByVal PinNumber As Byte, _
    ByVal Direction As Byte, _
    ByVal DutyCycle As Byte)

    ' Remember that output will be INVERTED, so invert everything here too.

    If (1 = Direction) Then 'When going forward, active-low output
        DutyCycle = Not DutyCycle
    End If

    ' Set the proper pin.
    If (PinNumber = LeftPWM) Then
        Call PutPin(LeftFront, Direction)
        Register.OCR1BH = 0
        Register.OCR1BL = DutyCycle
    ElseIf (PinNumber = RightPWM) Then
        Call PutPin(RightFront, Direction)
        Register.OCR1AH = 0
        Register.OCR1AL = DutyCycle
    End If
End Sub

'-----
Public Sub setPWMfloat( _
    ByVal PinNumber As Byte, _

```

```

    ByVal DutyCycle As Single)

' This procedure starts a PWM pulse train on the specified pin number. The
' nondimensional DutyCycle should be in range -1.0 to 1.0.

    Dim iDutyCycle As Byte
    Dim OtherValue As Byte ' the desired logic value of the motor's other pin

    ' Scale and handle direction
    ' Output is INVERTED, so we invert everything in this section too
    DutyCycle = -DutyCycle
    If (DutyCycle < 0.0) Then 'When going backwards, we become active-low
        iDutyCycle = FixB(((1.0 + DutyCycle) * 255.0) + 0.5) ' Round off.
        OtherValue = 1
    Else
        iDutyCycle = FixB((DutyCycle * 255.0) + 0.5) ' Round off.
        OtherValue = 0
    End If

    ' Set the proper pin.
    If (PinNumber = LeftPWM) Then
        Call PutPin(LeftFront, OtherValue)
        Register.OCR1BH = 0
        Register.OCR1BL = iDutyCycle
    ElseIf (PinNumber = RightPWM) Then
        Call PutPin(RightFront, OtherValue)
        Register.OCR1AH = 0
        Register.OCR1AL = iDutyCycle
    End If

End Sub

'-----
Public Sub InitializePWM( _
    ByVal RateSetting As Byte)

    ' RateSetting ranges from 1=14,456 Hz to 5=14.12 Hz for 8-bit PWM

    Const PWMmode8bit As Byte = bx0000_0001
    Const PWMmode9bit As Byte = bx0000_0010
    Const PWMmode10bit As Byte = bx0000_0011
    Const PWMmodeOff As Byte = bx0000_0000

    Const MaskOC1A As Byte = bx1000_0000
    Const MaskOC1B As Byte = bx0010_0000

    ' Turn off Timer1.
    Register.TCCR1B = 0

    ' Set Timer1 to 8-bit PWM mode.
    Register.TCCR1A = PWMmode8bit

    ' Initialize pin directions.
    Call PutPin(PinOC1A, bxOutputLow)
    Call PutPin(PinOC1B, bxOutputLow)

    ' Clear duty cycles on both OCR1A and OCR1B pins.
    Register.OCR1AH = 0
    Register.OCR1AL = 0

    Register.OCR1BH = 0
    Register.OCR1BL = 0

    ' Start Timer1 according to the specified rate setting.
    Register.TCCR1B = RateSetting

    ' Enable PWM for both pins.
    Register.TCCR1A = Register.TCCR1A Or MaskOC1A
    Register.TCCR1A = Register.TCCR1A Or MaskOC1B

End Sub

'-----
Public Sub PutPinPWM( _
    ByVal PinNumber As Byte, _
    ByVal DutyCycle As Single)

```



```

' This procedure starts a PWM pulse train on the specified pin number. The
' nondimensional DutyCycle should be in range 0.0 to 1.0.

    Dim iDutyCycle As Byte

    ' Scale and enforce range constraints.
    If (DutyCycle < 0.0) Then
        iDutyCycle = 0
    ElseIf (DutyCycle > 1.0) Then
        iDutyCycle = 255
    Else
        iDutyCycle = FixB((DutyCycle * 255.0) + 0.5) ' Round off.
    End If

    ' Set the proper pin.
    If (PinNumber = PinOC1A) Then
        Register.OCR1AH = 0
        Register.OCR1AL = iDutyCycle
    ElseIf (PinNumber = PinOC1B) Then
        Register.OCR1BH = 0
        Register.OCR1BL = iDutyCycle
    End If

End Sub
'-----
'-----
' Transfer data to and from the serial port.
'-----
Public Sub OpenSerialPort( _
    ByVal PortNumber As Byte, _
    ByVal BaudRate As Long)

' Opens a serial port at the specified baud rate.

' Com1 requires that the network be disabled. On the BasicX-01
' Developer Board, it may be necessary to raise pin 14, which can be
' done here or in the chip I/O initialization.
'>>If (PortNumber = 1) Then
'>>    Call PutPin(14, bxOutputHigh)
'>>End If

    Call OpenQueue(InputBuffer, InputBufferSize)

    Call OpenQueue(OutputBuffer, OutputBufferSize)

    Call OpenCom(PortNumber, BaudRate, InputBuffer, OutputBuffer)

End Sub
'-----
Public Sub GetByte( _
    ByRef Value As Byte, _
    ByRef Success As Boolean)

' Inputs a byte from the serial port, if available. Returns regardless. The
' Success flag is set depending on whether a byte is available.
'
' The byte is in direct binary format -- it is not in string format.

    ' Find out if anything is in the queue.
    Success = StatusQueue(InputBuffer)

    ' If data is in the queue, extract it.
    If (Success) Then
        Call GetQueue(InputBuffer, Value, 1)
    Else
        Value = 0
    End If

End Sub
'-----
Public Sub PutByte( _
    ByVal Value As Byte)

' Sends one byte of binary data to the serial port. The byte is sent
' directly without translating it to a string.

```

```

    Call PutQueue(OutputBuffer, Value, 1)
End Sub
'-----
Public Sub NewLine()
' Outputs a <CR> <LF> to the serial port.
'Our output will be processed by UNIX, so send only the LF!!!
'   Call PutByte(ASCII_CR)
'   Call PutByte(ASCII_LF)
End Sub
'-----
Public Sub PutLine( _
    ByRef Tx As String)
' Outputs a String type, followed by <CR> <LF>. Output is to the serial
' port.
    Call PutStr(Tx)
    Call NewLine
End Sub
'-----
Public Sub PutStr( _
    ByRef Tx As String)
' Outputs a String type to the serial port.
    Dim Length As Integer, Ch As String * 1, bCh As Byte
    Dim I As Integer
    Length = Len(Tx)
    For I = 1 To Length
        Ch = Mid(Tx, I, 1)
        bCh = Asc(Ch)
        Call PutByte(bCh)
    Next
End Sub
'-----
' Less frequently used output commands; try using Debug.Print instead
'-----
Public Sub PutB( _
    ByVal Value As Byte)
' Outputs a Byte type to the serial port.
    Dim Digit(1 To 3) As Byte
    Dim i As Integer, NDigits As Integer
    Const Base As Byte = 10
    NDigits = 0
    Do
        NDigits = NDigits + 1
        Digit(NDigits) = Value Mod Base
        Value = Value \ Base
    Loop Until (Value = 0)
    For i = NDigits To 1 Step -1
        Call PutByte(Digit(i) + ASCIIzero)
    Next
End Sub
'-----
Public Sub PutHexB( _
    ByVal Value As Byte)
' Outputs a Byte type to the serial port. Hexadecimal format is used.
    Dim Digit(1 To 2) As Byte, D As Byte
    Dim i As Integer, NDigits As Integer

```

```

Const Base As Byte = 16
Const ASCIIhexBias As Byte = 55

NDigits = 0

Do
    NDigits = NDigits + 1

    D = Value Mod Base
    If (D < 10) Then
        D = D + ASCIIzero
    Else
        D = D + ASCIIhexBias
    End If

    Digit(NDigits) = D

    Value = Value \ Base
Loop Until (Value = 0)

For i = NDigits To 1 Step -1
    Call PutByte(Digit(i))
Next

End Sub
'-----
Public Sub PutI( _
    ByVal Value As Integer)

' Outputs an Integer type to the serial port.

    Call PutL(CLng(Value))

End Sub
'-----
Public Sub PutUI( _
    ByVal Value As UnsignedInteger)

' Outputs an UnsignedInteger type to the serial port.

    Dim L As Long, Tmp As New UnsignedInteger

    Tmp = Value

    L = 0

    ' Copy Value into the lower two bytes of L.
    Call BlockMove(2, MemAddress(Tmp), MemAddress(L))

    Call PutL(L)

End Sub
'-----
Public Sub PutUL( _
    ByVal Value As UnsignedLong)

' Outputs an UnsignedLong type to the serial port.

    Dim UL As New UnsignedLong, L As Long, Digit As New UnsignedLong
    Dim I As Integer, Temp As New UnsignedLong

    ' If the top bit is clear, the number is ready to go.
    If ((Value And &H80000000) = 0) Then
        Call PutL(CLng(Value))
        Exit Sub
    End If

    ' Divide by 10 is done by a right shift followed by a divide by 5.
    ' First clear top bit so we can do a signed divide.
    UL = Value
    UL = UL And &H7FFFFFFF

    ' Shift to the right 1 bit.
    L = CLng(UL)
    L = L \ 2

    ' Put the top bit back, except shifted to the right 1 bit.

```

```

    UL = CuLng(L)
    UL = UL Or &H40000000

    ' The number now fits in a signed long.
    L = CLng(UL)

    L = L \ 5

    Call PutL(L)

    ' Multiply by 10. Since multiply is not implemented for UnsignedLong, we
    ' do the equivalent addition.
    Temp = CuLng(L)
    UL = 0
    For I = 1 To 10
        UL = UL + Temp
    Next

    ' Find the rightmost digit.
    Digit = Value - UL
    Call PutL(CLng(Digit))

End Sub
'-----
Public Sub PutL( _
    ByVal Value As Long)

    ' Outputs a Long type to the serial port.

    ' Reserve space for "2147483648".
    Dim Digit(1 To 10) As Byte
    Dim NDigits As Integer
    Dim i As Integer
    Const Base As Long = 10

    ' The working number must be zero or negative. Otherwise the negative
    ' limit will cause overflow if we take its absolute value.
    If (Value < 0) Then
        Call PutByte(ASCIIMinus)
    Else
        Value = -Value
    End If

    NDigits = 0

    Do
        NDigits = NDigits + 1
        Digit(NDigits) = CByte( Abs(Value Mod Base) )
        Value = Value \ Base
    Loop Until (Value = 0)

    ' Digits are stored in reverse order of display.
    For i = NDigits To 1 Step -1
        Call PutByte(Digit(i) + ASCIIzero)
    Next

End Sub
'-----
Public Sub PutSci( _
    ByVal Value As Single)

    ' Outputs floating point number in scientific notation format. The format
    ' is such that 13 characters are always generated. Sign characters are
    ' included for both mantissa and exponent. Exponents have 2 digits,
    ' including a leading zero if necessary.
    '
    ' Example Formats: "+1.234567E+00"
    '                  "-7.654321E-20"
    '                  "+3.141593E+05"
    '                  "+0.000000E+00"

    Dim Mantissa As Single, Exponent As Integer, LMant As Long

    Call SplitFloat(Value, Mantissa, Exponent)

    ' Sign.
    If (Mantissa < 0.0) Then

```

```

        Call PutByte(ASCIIminus)
    Else
        Call PutByte(ASCIIplus)
    End If

    ' Convert mantissa to a 7-digit integer.
    LMant = FixL((Abs(Mantissa) * 1000000.0) + 0.5)

    ' Correct for roundoff error. Mantissa can't be > 9.999999
    If (LMant > 9999999) Then
        LMant = 9999999
    End If

    ' First digit of mantissa.
    Call PutByte( CByte(LMant \ 1000000) + ASCIIzero)

    ' Decimal point.
    Call PutByte(ASCIIdecimal)

    ' Remaining digits of mantissa.
    LMant = LMant Mod 1000000

    Call InsertZeros(LMant)

    Call PutL(LMant)

    ' Exponent.
    Call PutByte(69) ' E

    If (Exponent < 0) Then
        Call PutByte(ASCIIminus)
    Else
        Call PutByte(ASCIIplus)
    End If

    ' A 2-digit exponent has a leading zero.
    If (Abs(Exponent) < 10) Then
        Call PutByte(ASCIIzero)
    End If

    Call PutI(Abs(Exponent))

End Sub
'-----
Private Sub InsertZeros( _
    ByVal X As Long)

    Dim NumZeros As Byte, I As Byte

    If (X >= 100000) Then
        Exit Sub ' 100 000 <= X
    ElseIf (X >= 10000) Then
        NumZeros = 1 ' 10 000 <= X <= 99 999
    ElseIf (X >= 1000) Then
        NumZeros = 2 ' 1 000 <= X <= 9 999
    ElseIf (X >= 100) Then
        NumZeros = 3 ' 100 <= X <= 999
    ElseIf (X >= 10) Then
        NumZeros = 4 ' 10 <= X <= 99
    Else
        NumZeros = 5 ' 0 <= X <= 9
    End If

    For I = 1 To NumZeros
        Call PutByte(ASCIIzero)
    Next

End Sub
'-----
Public Sub PutS( _
    ByVal Value As Single)

    ' Outputs a floating point number to the serial port. If the number can be
    ' displayed without using scientific notation, it is. Otherwise scientific
    ' notation is used.

    Dim X As Single, DecimalPlace As Integer, Mantissa As Single

```

```

Dim Exponent As Integer, DigitPosition As Integer, Factor As Long
Dim LMant As Long, DecimalHasDisplayed As Boolean

' Special case for zero.
If (Value = 0.0) Then
    Call PutByte(ASCIIzero)
    Call PutByte(ASCIIdecimal)
    Call PutByte(ASCIIzero)
    Exit Sub
End If

X = Abs(Value)

' Use scientific notation for values too big or too small.
If (X < 0.1) Or (X > 999999.9) Then
    Call PutSci(Value)
    Exit Sub
End If

' What follows is non-exponent displays for 0.1000000 < Value < 999999.9

' Sign.
If (Value < 0.0) Then
    Call PutByte(ASCIIminus)
End If

If (X < 1.0) Then
    Call PutByte(ASCIIzero) ' Leading zero.
    Call PutByte(ASCIIdecimal)
    DecimalHasDisplayed = True
    DecimalPlace = 0

    ' Convert number to a 7-digit integer.
    LMant = FixL((X * 10000000.0) + 0.5)
Else
    Call SplitFloat(X, Mantissa, Exponent)
    DecimalPlace = Exponent + 2

    ' Convert mantissa to a 7-digit integer.
    LMant = FixL((Abs(Mantissa) * 1000000.0) + 0.5)

    ' Correct for roundoff error. Mantissa can't be > 9.999999
    If (LMant > 9999999) Then
        LMant = 9999999
    End If

    DecimalHasDisplayed = False
End If

Factor = 1000000

For DigitPosition = 1 To 7

    If (DigitPosition = DecimalPlace) Then
        Call PutByte(ASCIIdecimal)
        DecimalHasDisplayed = True
    End If

    Call PutByte( CByte(LMant \ Factor) + ASCIIzero )

    LMant = LMant Mod Factor

    ' Stop trailing zeros, except for one immediately following the
    ' decimal place.
    If (LMant = 0) Then
        If (DecimalHasDisplayed) Then
            Exit Sub
        End If
    End If

    Factor = Factor \ 10
Next

End Sub
'-----
Private Sub SplitFloat( _
    ByVal Value As Single, _

```

```

ByRef Mantissa As Single, _
ByRef Exponent As Integer) _
' Splits a floating point number into mantissa and exponent. The mantissa
' range is such that 1.0 <= Abs(Mantissa) < 10.0 for nonzero numbers, and
' zero otherwise.

Dim X As Single, Factor As Single

' Zero is a special case.
If (Value = 0.0) Then
    Mantissa = 0.0
    Exponent = 0
    Exit Sub
End If

X = Abs(Value)

Exponent = 0
Factor = 1.0

' Multiply or divide by ten to transform number to value between 1 and 10.
Do
    If (X >= 10.0) Then
        X = X / 10.0
        Factor = Factor * 10.0
        Exponent = Exponent + 1
    ElseIf (X < 1.0) Then
        X = X * 10.0
        Factor = Factor * 10.0
        Exponent = Exponent - 1
    Else
        ' When we reach this point, then 1.0 <= mantissa < 10.0.
        Exit Do
    End If
Loop

' Determine mantissa.
If (Exponent = 0) Then
    Mantissa = Value
ElseIf (Exponent > 0) Then
    Mantissa = Value / Factor
Else
    Mantissa = Value * Factor
End If

End Sub
'-----

```

Section 8.3.2 Infinite Atom Linux Distribution

Section 8.3.2.1 Description and Explanation

Linux is an advanced open-source UNIX-like operating system available for numerous hardware architectures. The nature of Linux is described in Section 4.4.3.2 and an overview of Infinite Atom containing some important details is provided in Section 4.4.3.3. The complete contents of the EvBot's Infinite Atom Linux distribution (except for MATLAB, which is commercial software) are included on the CD-ROM as tarred and gzipped archives in the "EvBot Software\Infinite Atom\Filesystem contents"

directory. Tar and gzip were used instead of the more popular “zip” format in order to preserve the security attributes of Infinite Atom’s files. It is possible for those without access to tar and gzip to use a recent version of the popular WinZip program (www.winzip.com) instead of tar and gzip to read the archives.

Section 8.3.2.1.1 Basic EvBot Control Commands

There are four basic commands included as part of Infinite Atom that were developed just for the EvBot. They are used to start and stop an EvBot’s MATLAB controller and toggle the ATA flash card’s filesystem between read-only and read/write modes. They were specially developed to allow the normal “robot” EvBot login ID to perform common actions normally only possible for the high-security root login ID.

Infinite Atom is preconfigured to automatically start an EvBot’s MATLAB controller on bootup and to start with the ATA flash card’s filesystem in read-only mode to prevent data corruption should the EvBot’s battery die during operation. The controller can be stopped without shutting down the EvBot by logging onto the EvBot over the wireless network using SSH and then running the command `stopautodrive`, which stops MATLAB (and hence the MATLAB controller) and remounts the ATA flash card’s filesystem read/write so that the MATLAB controller can be easily modified. After modifications have been made, the MATLAB controller can be restarted with the `startautodrive` command which changes the ATA flash card’s filesystem back to read-only mode and then restarts MATLAB. The ATA flash card’s filesystem can also be toggled between the read-only and read/write modes by using the `ro` and `rw` commands, respectively. Prolonged operation of the EvBot under battery power with the ATA flash

card's filesystem in read/write mode is discouraged. Complete source code for these four commands is available on the CD-ROM in the "EvBot Software\Infinite Atom\Special nonstandard components" directory. They should be built using the standard make command.

Section 8.3.2.1.2 Ram Disk for Temporary Storage

Another special feature of Infinite Atom mentioned in Section 4.4.3.3 is that it uses a 4 MB ram disk to hold all non-permanent files. Its use allows both the DOC and the ATA flash card to be mounted read-only while still maintaining the abilities to keep log files, temporarily record data, and pass data between programs using temporary files. The ram disk is mounted as the /var filesystem, but also holds the contents of /tmp (which is a symbolic link to /var/tmp). Because the /var filesystem has an established directory hierarchy and the contents of a ram disk are not preserved after EvBot shutdown, /etc/rc.d/rc.sysinit must create the /var directory hierarchy after creating the ram disk on every bootup.

Section 8.3.2.1.3 Linux Startup Sequence Customizations

Despite the broad hardware support of Linux, there were serious difficulties getting Linux to boot properly from the DOC. Only the newest 2.4.x series of kernels had proper support for USB web cams, which necessitated their use. Unfortunately, the boot process was changed for the 2.4.x series and apparently was not fully debugged when running on "abnormal" x86 hardware, where "abnormal" includes such things as using a DOC as the boot device. Unlike Windows-based operating systems, Linux does

not (and cannot) use the motherboard's onboard Basic Input/Output System (BIOS) for disk access because all standards-compliant BIOS implementations are inherently slow by design. Since the DOC masquerades itself as a disk by loading BIOS extensions, Linux is unaffected and hence does not naturally recognize the DOC. M-Systems (<http://www.m-sys.com/>), maker of the DOC, provides a patch to the Linux kernel source code to make Linux recognize and use the DOC. To preserve their intellectual property, M-Systems has chosen to make part of the patch only available as a binary object file. Because Linux is licensed as open-source under the terms of the GNU project's General Public License (GPL), it is legally forbidden to ever distribute a Linux kernel that includes any object code for which the source code is not publicly available. Therefore, the only way a Linux kernel capable of accessing a DOC can be distributed or otherwise made publicly available is by compiling the DOC support as a loadable kernel module, keeping the closed-source code out of the actual kernel. The initial problem is that to boot off the DOC, the kernel must load the DOC module from somewhere before accessing the DOC. This problem has only one solution: an initial RAM disk.

Linux is typically loaded into memory by another program (called a "Linux loader") before being given complete control of the computer. Once the Linux kernel is running, it starts the init program to supervise system startup (see Figure 8.29). Linux provides a mechanism for the Linux loader to preload the image (or copy) of a RAM disk that the kernel can access as a disk drive before any hardware disks become available to the kernel. By placing the DOC kernel module and startup code to load it (named `linuxrc`) on the image used as the initial RAM disk, the Linux kernel can startup, access

the initial RAM disk, load the DOC module, and then—theoretically—replace the RAM disk with the DOC as the root filesystem before starting init from the DOC. Though somewhat complex, it is relatively straight-forward to set up a system to boot as described. The real difficulty in booting Linux on the EvBot was that the 2.4.x series of kernels use new code to handle the initial RAM disk, and the new code is broken when using the DOC instead of a real disk drive as the root filesystem. Several customizations were necessary to make Linux boot from the DOC, some of which were expected, but others were the result of the kernel (or possibly the DOC driver) not working properly.

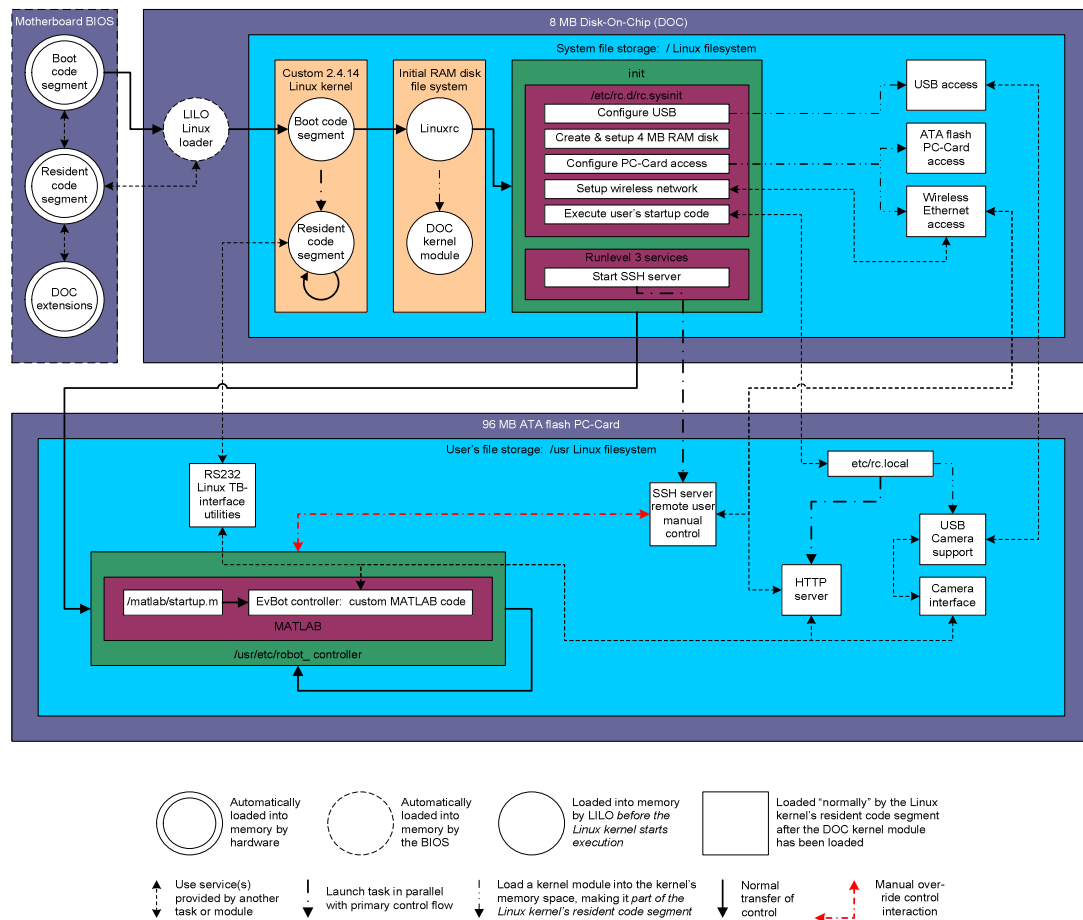


Figure 8.29 PC/104 stack high-level flowchart for the boot sequence

Section 8.3.2.1.4 Modifications to LILO

The EvBot boot process begins with the BIOS loading the Linux loader off of the DOC. The Linux loader used for the EvBot is a version of LILO (short for Linux Loader) called dLILO (for DOC LILO) that has been modified to be able to read from the DOC using the BIOS. DLILO reads the kernel and the initial RAM disk image from the DOC, loads them into memory, and then transfers control to the kernel after passing it certain bootup arguments, one of which tells the kernel to send its system messages to the second serial port instead of to a video card, since EvBots normally lack a video card.

Section 8.3.2.1.5 Modifications to the Kernel

Several modifications to the Linux kernel were necessary to make the EvBot work. When the EvBot project was begun, Phillips had not yet released Linux drivers for their web cams, and so the best supported USB web cams were the ones using the OV511 chipset. Despite their superior ranking, the quality of support was still lacking and so the latest OV511 drivers were patched into the kernel source code to make OV511 cameras perform better. (It was originally assumed that the OV511 problems were caused by the USB OHCI chipset Linux driver, because most error messages were printed by the OHCI driver.) There is also a bug in Linux or the MZ104 (or both) causing PnP configuration to not work correctly for PC-Card devices. The two serial ports on the MZ104 are not able to share their IRQ's with other devices, but Linux believes that they can and assigns one or both of them to PC-Card devices, preventing one or both serial ports from working after the PC-Cards are initialized. Since the TB is controlled using a serial port, this creates a serious problem that was ultimately solved by patching the PC/104 PC-Card

adapter's Linux driver to never use the IRQ's normally assigned to the two serial ports. It was also discovered that Linux incorrectly assigns IRQ 9 to PC-Card devices when no video card is installed, instantaneously locking the machine. Patching the PC/104 PC-Card adapter driver to never use IRQ 9 solved that problem as well. The final and most important modification made to the kernel was to patch it to use the DOC driver module.

Section 8.3.2.1.6 Redesign of the Initial RAM Disk

The most difficult task in designing and producing a working EvBot was making Linux correctly load an initial RAM disk and then correctly “unload” it to boot from the DOC. During normal startup, a Linux kernel mounts the root filesystem and then starts an init program, which is the one and only program the kernel starts itself. (Technically, a modern kernel also starts other processes, but they are unimportant to the current discussion.) Normally, init is a special program designed to setup the computer, start the login processes, and keep watch over running applications. However, since any program can be run as init by the kernel, all that is technically required to run a fully functional, statically compiled, stand-alone Linux application is the kernel. This fact is critical to the use of an initial RAM disk, which must provide only the bare essentials to reduce its size.

Numerous RAM-disk boot procedures were tried, each progressively more complex and drastic than the one before, and they all failed to successfully boot Linux from the DOC. Finally, after examining the kernel boot code for a recent (2.4.9) kernel and isolating the problem to the last phase of the boot sequence, a special init program (named linuxrc) was created for the initial RAM disk that used Linux system calls to completely bypass the last phase of the boot procedure and manually transfer control to

the DOC. Complete source code for `linuxrc` is included in Section 8.3.2.2. In order to bypass the last phase of the initial RAM disk boot procedure, the Linux kernel is told to use the initial RAM disk as a regular RAM disk for the root filesystem—that is, that the RAM disk is the final root device and the kernel should not try to change it. The kernel is also told to use `linuxrc` as the init process.

`Linuxrc` begins by loading the DOC module by calling a statically linked version of the `insmod` program. Because `linuxrc` and `insmod` are statically compiled, they need no other programs or libraries to run. `Linuxrc` then mounts the DOC to a subdirectory before using the `pivot_root` system call to move the DOC's filesystem to the root position in the filesystem tree and the old root device's filesystem (on the initial RAM disk) to a subdirectory. (Later the initial RAM disk is unmounted altogether, and it should not be confused with the writeable RAM disk used to save images and other data.) Finally, `linuxrc` *replaces* itself with the `init` program located on the DOC. It is important that the real `init` program replace `linuxrc` rather than just be started by `linuxrc` because `init` must run with process ID 0 in order to function correctly, and `linuxrc` has process ID 0. Actually, `linuxrc` does not directly replace itself with `init`, but rather it replaces itself with a special `startinit` BASH shell script that in turn replaces itself with `init`. The additional level of indirection is necessary because `linuxrc` was originally started from the initial RAM disk and so its input/output streams are tied to the device entries located on the initial RAM disk. When Linux replaces one program with another, it normally preserves the input/output streams, but to function correctly `init` must use input/output streams tied to the device entries on the DOC. When `startinit` replaces itself with `init`, it uses BASH to

forcefully redirect the input/output streams of init to the DOC; linuxrc is unable to do this because BASH is not available when linuxrc is started.

Section 8.3.2.1.7 Modification of the Init Scripts

Once the real init program is started, it parses the `/etc/inittab` file (included below in Section 8.3.2.3) to see which programs should be executed during startup and which ones should be continually running thereafter. As with a normal Red Hat distribution, `inittab` instructs `init` to run the script file `/etc/rc.d/rc.sysinit` on startup. The `rc.sysinit` script (included below in Section 8.3.2.4) is responsible for starting user-mode system services, configuring networking, loading device drivers, and otherwise preparing Infinite Atom to run MATLAB and other user programs, as shown in Figure 8.29. Like many of the EvBot's configuration files located in the `/etc` directory hierarchy, it has been heavily modified from the Red Hat original. It starts only services installed on the EvBot, it loads PC-Card support before such would normally be done so that the ATA Flash Card can be accessed for startup activities, and it configures some of the EvBot's custom hardware. One of the last things `rc.sysinit` does is to check for an `etc/rc.local` script on the ATA Flash Card's `/usr` filesystem and execute it if present. Users can thus easily add their own Linux startup code to Infinite Atom without having to modify the primary startup scripts. This is currently how the web server is started and the web cam drivers are loaded, as shown in Section 8.3.2.5, which lists the contents of `/usr/etc/rc.local`. As Section 8.3.2.3 shows, the `inittab` file has, however, been modified from Red Hat's version to not start graphical or console text mode login sessions, but instead to start a login process on the

second serial port and to keep the robot's controller (by default MATLAB with its output logged to the writeable RAM disk) continually running.

Section 8.3.2.1.8 MATLAB Customizations

Because MATLAB is commercial software, it has been omitted from the copy of the EvBot's filesystem contents included on the CD-ROM, but it can be easily added by anyone licensed to install MATLAB. In order to minimize MATLAB's required space, it was modified from its normal minimal install as described in Section 4.4.3.3.3. Specifically, to reduce storage use, the following files and directories were removed from MATLAB's installation directory: bin/matlabdoc, extern/examples, hpux10.pch, install, installguide_a4.ps, installguide.ps, install_matlab, install_matlabp.out, java, license.txt, matlabdoc, matlabdoc.html, opengl, setup.txt, startdoc, sunos5.pch, sys/ghostscript, sys/opengl, update, X11.

Section 8.3.2.2 Linuxrc C Source Code

```

/*****\
 * linuxrc -- handle initrd bootup, root FS mount, and init startup *
 * John M. Galeotti 6-11-01 *
 * *
 * Written to boot PC104 based small robots that use a DiskOnChip FS. *
 * Funded by DARPA *
\*****/

#include <stdio.h>
#include <unistd.h>
#include <sys/mount.h>
/* We need the pivot_root system call: */
#include "/usr/src/linux-2.4.5/include/linux/unistd.h"

static char * argv_init[10] = { "init", "3", NULL, };
static char * envp_init[12] = { "HOME=/", "TERM=linux", "BOOT_IMAGE=linux", "BOOT_FILE=/boot/bzImage",
NULL, };

int main (int argc, char *argv[]){
    int ps, tmp;

    printf("\n***** Running linuxrc *****\nlinuxrc ignores all arguments: ");
    for(tmp=0;tmp<argc;tmp++)
        printf("\n%s\n ", argv[tmp]);
}

```



```

printf("\nInstalling the DiskOnChip module:\n");
ps = fork();
if(ps == 0){
    execl("/bin/insmod", "/bin/insmod", "-v", "/lib/doc.o", NULL);
    exit(0);
}
waitpid(ps, &tmp, 0);

printf("Mounting /dev/msys/fla1 on /rootfs: ");
if(mount("/dev/msys/fla1", "/rootfs", "ext2", 0, NULL) != 0){
    printf("\n\n--> Error: Can't mount /dev/msys/fla1 <--\n\n");
    return (1); /* We have no root--Time to Die */
}
printf("done.\n");

/* Change the root FS with pivot_root */
chdir("/rootfs"); /* Make sure our CWD is on the new root fs */
printf("Changing /rootfs to / (old / is available in /initrd)\n");
if(pivot_root(".", "/initrd") != 0){
    printf("\n\n--> Error: pivot_root(/rootfs,/rootfs/initrd) failed! <--\n\n");
    return (1); /* pivot_root failed--Time to Die */
}
chdir("/"); /* Make sure our CWD is still on the new root fs */

printf("***** Running /etc/startinit *****\n\n");

execve("/etc/startinit",argv_init,envp_init);

/* We should never reach this line */
printf("\n\nBIG UH-OH! Something went wrong calling /etc/startinit\n");
return(0);
}

```

Section 8.3.2.3 /etc/inittab

```

#

# inittab          This file describes how the INIT process should set up
#                  the system in a certain run-level.
#
# Author:          Miquel van Smoorenburg, <miquels@drinkel.nl.mugnet.org>
#                  Modified for RHS Linux by Marc Ewing and Donnie Barnes
#

# Default runlevel. The runlevels used by RHS are:
# 0 - halt (Do NOT set initdefault to this)
# 1 - Single user mode
# 2 - Multiuser, without NFS (The same as 3, if you do not have networking)
# 3 - Full multiuser mode

# 4 - unused
# 5 - X11
# 6 - reboot (Do NOT set initdefault to this)
#
id:3:initdefault:

ok:3:once:/etc/ok_bEEP

# System initialization.
si::sysinit:/etc/rc.d/rc.sysinit

10:0:wait:/etc/rc.d/rc 0
11:1:wait:/etc/rc.d/rc 1
12:2:wait:/etc/rc.d/rc 2
13:3:wait:/etc/rc.d/rc 3
14:4:wait:/etc/rc.d/rc 4
15:5:wait:/etc/rc.d/rc 5
16:6:wait:/etc/rc.d/rc 6

# Things to run in every runlevel.
ud::once:/sbin/update

# Trap CTRL-ALT-DELETE
ca::ctrlaltdel:/sbin/shutdown -t3 -r now

```

```

# When our UPS tells us power has failed, assume we have a few minutes
# of power left. Schedule a shutdown for 2 minutes from now.
# This does, of course, assume you have powerd installed and your
# UPS connected and working correctly.
pf::powerfail:/sbin/shutdown -f -h +2 "Power Failure; System Shutting Down"

# If power was restored before the shutdown kicked in, cancel it.
pr:12345:powerokwait:/sbin/shutdown -c "Power Restored; Shutdown Cancelled"

# Run Matlab Control in the default runlevel only

m:3:respawn:/usr/etc/robot_controller

# Run gettys in standard runlevels
#1:2345:respawn:/sbin/mingetty --noclear tty1
#2:2345:respawn:/sbin/mingetty tty2
#3:2345:respawn:/sbin/mingetty tty3
#4:2345:respawn:/sbin/mingetty tty4
#5:2345:respawn:/sbin/mingetty tty5
#6:2345:respawn:/sbin/mingetty tty6
s:235:respawn:/sbin/agetty ttyS1 19200

# DO NOT Run xdm in runlevel 5, BECAUSE WE ARE A SMALL ROBOT
# xdm is now a separate service
# x:5:respawn:/etc/X11/prefdm -nodaemon

```

Section 8.3.2.4 /etc/rc.d/rc.sysinit

```

#!/bin/bash

#
# /etc/rc.sysinit - run once at boot time
#
# Taken in part from Miquel van Smoorenburg's bcheckrc.
#

# Rerun ourselves through initlog
if [ -z "$IN_INITLOG" ]; then
    [ -f /sbin/initlog ] && exec /sbin/initlog $INITLOG_ARGS -r /etc/rc.sysinit
fi

# If we're using devfs, start devfsd now - we need the old device names
[ -e /dev/.devfsd -a -x /sbin/devfsd ] && /sbin/devfsd /dev

# Set the path
PATH=/bin:/sbin:/usr/bin:/usr/sbin
export PATH

HOSTNAME=`/bin/hostname`

# Read in config data.
if [ -f /etc/sysconfig/network ]; then
    . /etc/sysconfig/network
else
    NETWORKING=no
fi

if [ -z "$HOSTNAME" -o "$HOSTNAME" = "(none)" ]; then
    HOSTNAME=localhost
fi

# Source functions
. /etc/init.d/functions

# Print a banner. ;)
echo -en $"\\t\\t    Welcome to "
[ "$BOOTUP" != "serial" ] && echo -en $"\\033[1;31m"
echo -en $"Infinite Atom"
[ "$BOOTUP" != "serial" ] && echo -en $"\\033[0;39m"
echo $" Linux"
if [ "$PROMPT" != "no" ]; then
    echo -en $"\\t\\tPress 'I' to enter interactive startup."
    echo
    sleep 1
fi

```

```

# Fix console loglevel
/bin/dmesg -n $LOGLEVEL

# Mount /proc (done here so volume labels can work with fsck)

action $"Mounting proc filesystem: " mount -n -t proc /proc /proc

echo $"`grep flal /proc/mounts`"

# Configure kernel parameters

action $"Configuring kernel parameters: " sysctl -e -p /etc/sysctl.conf

# Set the system clock.
ARC=0
SRM=0
UTC=0

if [ -f /etc/sysconfig/clock ]; then
    . /etc/sysconfig/clock

    # convert old style clock config to new values
    if [ "${CLOCKMODE}" = "GMT" ]; then
        UTC=true
    elif [ "${CLOCKMODE}" = "ARC" ]; then
        ARC=true
    fi
fi

CLOCKDEF=""
CLOCKFLAGS="--hctosys"

case "$UTC" in
    yes|true)
        CLOCKFLAGS="$CLOCKFLAGS -u";
        CLOCKDEF="$CLOCKDEF (utc)";
        ;;
    no|false)
        CLOCKFLAGS="$CLOCKFLAGS --localtime";
        CLOCKDEF="$CLOCKDEF (localtime)";
        ;;
esac

case "$ARC" in
    yes|true)
        CLOCKFLAGS="$CLOCKFLAGS -A";
        CLOCKDEF="$CLOCKDEF (arc)";
        ;;
esac

case "$SRM" in
    yes|true)
        CLOCKFLAGS="$CLOCKFLAGS -S";
        CLOCKDEF="$CLOCKDEF (srm)";
        ;;
esac

/sbin/hwclock $CLOCKFLAGS

action $"Setting clock $CLOCKDEF: `date`" date

# Set the hostname.
action $"Setting hostname ${HOSTNAME}: " hostname ${HOSTNAME}

# Initialize USB controller and HID devices
usb=0
if ! grep -iq "nousb" /proc/cmdline 2>/dev/null && ! grep -q "usb" /proc/devices 2>/dev/null ; then
    aliases=~ /sbin/modprobe -c | egrep -s "^alias[[:space:]]+usb-controller" | awk '{ print $3 }'`
    if [ -n "$aliases" -a "$aliases" != "off" ] ; then
        modprobe usbcore
        action $"Mounting USB filesystem: " mount -n -t usbdevfs usbdevfs /proc/bus/usb
        for alias in $aliases ; do
            action $"Initializing USB controller ($alias): " modprobe $alias
        done
        [ $? -eq 0 -a -n "$aliases" ] && usb=1
    fi
fi

```

```

if ! grep -iq "nousb" /proc/cmdline 2>/dev/null && grep -q "usb" /proc/devices 2>/dev/null ; then
    usb=1
fi

needusbstorage=
if [ $usb = "1" ]; then
    sleep 5
    mouseoutput=`cat /proc/bus/usb/devices 2>/dev/null|grep -E "^I.*Cls=03.*Prot=02"``
    kbddoutput=`cat /proc/bus/usb/devices 2>/dev/null|grep -E "^I.*Cls=03.*Prot=01"``
    needusbstorage=`cat /proc/bus/usb/devices 2>/dev/null|grep -e "^I.*Cls=08"``
    if [ -n "$kbddoutput" ] || [ -n "$mouseoutput" ]; then
        action $"Initializing USB HID interface: " modprobe hid 2> /dev/null
    fi
    if [ -n "$kbddoutput" ]; then
        action $"Initializing USB keyboard: " modprobe keybdev
    fi
    if [ -n "$mouseoutput" ]; then
        action $"Initializing USB mouse: " modprobe mousedev
    fi
fi

ROOTFSTYPE=`grep " / " /proc/mounts | awk '{ print $3 }'``
echo $"Warning: Infinite Atom does not check file systems on startup"

# check for arguments

if grep -iq nopnp /proc/cmdline >/dev/null 2>&1 ; then
    PNP=
else
    PNP=yes
fi

# Remount the root filesystem read-write.
#state=`awk '/(^\/dev\/root| \\/ )/ { print $4 }' /proc/mounts`
#[ "$state" != "rw" ] && \
# action $"Remounting root filesystem in read-write mode: " mount -n -o remount,rw /

# Clear mtab
#>/etc/mtab

# Remove stale backups
#rm -f /etc/mtab~ /etc/mtab~~

# Enter root, /proc and (potentially) /proc/bus/usb and devfs into mtab.
#mount -f /
#mount -f /proc
[ -f /proc/bus/usb/devices ] && mount -n -f -t usbdevfs usbdevfs /proc/bus/usb
[ -e /dev/.devfsd ] && mount -n -f -t devfs devfs /dev

# The root filesystem is now read-write, so we can now log via syslog() directly..
if [ -n "$IN_INITLOG" ]; then
    IN_INITLOG=
fi

if ! grep -iq nomodules /proc/cmdline >/dev/null 2>&1 && [ -f /proc/ksyms ]; then
    USEMODULES=y
else
    USEMODULES=
fi

if [ -x /sbin/depmod -a -n "$USEMODULES" ]; then
    # If they aren't using a recent sane kernel, make a link for them
    if [ ! -n "`uname -r | grep -- "-"`" ]; then
        ktag=`cat /proc/version`
        mtag=`grep -l "$ktag" /lib/modules/*/.rhmvtag 2> /dev/null`
        if [ -n "$mtag" ]; then
            mver=`echo $mtag | sed -e 's,/lib/modules/,,' -e 's,/.rhmvtag,, ' -e 's,[ ]*$,,`
            fi
            if [ -n "$mver" ]; then
                ln -sf /lib/modules/$mver /lib/modules/default
            fi
        fi
        if [ -L /lib/modules/default ]; then
            INITLOG_ARGS= action $"Finding module dependencies: " depmod -A default
        else
            INITLOG_ARGS= action $"Finding module dependencies: " depmod -A
        fi
    fi
fi

```

```

    fi
fi

if [ -f /proc/sys/kernel/modprobe ]; then
    if [ -n "$USEMODULES" ]; then
        sysctl -w kernel.modprobe="/sbin/modprobe" >/dev/null 2>&1
        sysctl -w kernel.hotplug="/sbin/hotplug" >/dev/null 2>&1
    else
        # We used to set this to NULL, but that causes 'failed to exec' messages"
        sysctl -w kernel.modprobe="/bin/true" >/dev/null 2>&1
        sysctl -w kernel.hotplug="/bin/true" >/dev/null 2>&1
    fi
fi

# Load modules (for backward compatibility with VARs)
if [ -f /etc/rc.modules ]; then
    /etc/rc.modules
fi

action $"Preparing the /var ramdisk filesystem: " /sbin/mkfs.minix /dev/ramdisk_var_fs

# Mount all other filesystems (except for NFS and /proc, which is already
# mounted). Contrary to standard usage,
# filesystems are NOT unmounted in single user mode.

# For some reason mount returns an error code, even though it seems to work...
#action $"Mounting local filesystems: " mount -n -a -t nonfs,smbfs,ncpfs
action $"Mounting local filesystems: " /bin/true
    mount -n -a -t nonfs,smbfs,ncpfs > /dev/console 2>/dev/console

cd /var
action $"Creating the /var directory heirarchy: " mkdir log run lib state state/pcmcia lock
lock/subsys lock/console tmp && chmod 1777 tmp
cd /

action $"Unmounting /initrd: " umount /initrd

# Clean out /.
rm -f /fastboot /fsckoptions /forcefsck /halt /poweroff

# Do we need (w|u)tmpx files? We don't set them up, but the sysadmin might...
_NEED_XFILES=
[ -f /var/run/utmpx -o -f /var/log/wtmpx ] && _NEED_XFILES=1

action $"Starting Card Services: " /sbin/cardmgr
sleep 6
action $"Mounting ATA-Flash-Card: " mount -n -o ro /dev/hde1 /usr
action $"Setting up Network: " /sbin/ifup eth0 && /etc/pcmcia/wireless start eth0

# Wait for DHCP to do its magic
sleep 3

# We use pump for DHCP, but pump doesn't set the hostname correctly--fix it:
IPADDR=`LANG= LC_ALL= ifconfig eth0 | grep 'inet addr' |
    awk -F: '{ print $2 } ' | awk '{ print $1 }'`
eval `~/bin/ipcalc --silent --hostname ${IPADDR}`
if [ "$?" = "0" ]; then
    hostname $HOSTNAME
fi

action $"Remounting Root filesystem Read Only: " mount -o remount,ro /

# Reset pam_console permissions
[ -x /sbin/pam_console_apply ] && /sbin/pam_console_apply -r

{
# Clean up utmp/wtmp
>/var/run/utmp
touch /var/log/wtmp
chgrp utmp /var/run/utmp /var/log/wtmp
chmod 0664 /var/run/utmp /var/log/wtmp
}

```

```

if [ -n "$_NEED_XFILES" ]; then
    >/var/run/utmpx
    touch /var/log/wtmpx
    chgrp utmp /var/run/utmpx /var/log/wtmpx
    chmod 0664 /var/run/utmpx /var/log/wtmpx
fi

# Initialize the serial ports.
if [ -f /etc/rc.serial ]; then
    . /etc/rc.serial
fi

# Load usb storage here, to match most other things
if [ -n "$_NEEDUSBSTORAGE" ]; then
    modprobe usb-storage >/dev/null 2>&1
fi

# If they asked for ide-scsi, load it
if grep -q "ide-scsi" /proc/cmdline ; then
    modprobe ide-cd >/dev/null 2>&1
    modprobe ide-scsi >/dev/null 2>&1
fi

# Generate a header that defines the boot kernel.
#/sbin/mkkerneldoth

# Adjust symlinks as necessary in /boot to keep system services from
# spewing messages about mismatched System maps and so on.
if [ -L /boot/System.map -a -r /boot/System.map-`uname -r` ] ; then
    ln -s -f System.map-`uname -r` /boot/System.map
fi
if [ ! -e /boot/System.map -a -r /boot/System.map-`uname -r` ] ; then
    ln -s -f System.map-`uname -r` /boot/System.map
fi

# Perform any of the user's startup actions
if [ -f /usr/etc/rc.local ] ; then
    . /usr/etc/rc.local
fi

# Now that we have all of our basic modules loaded and the kernel going,
# let's dump the syslog ring somewhere so we can find it later
dmesg > /var/log/dmesg
sleep 1
kill -TERM `sbin/pidof getkey` >/dev/null 2>&1
} &
if [ "$PROMPT" != "no" ]; then
    /sbin/getkey i && touch /var/run/confirm
fi

wait

```

Section 8.3.2.5 /usr/etc/rc.local

```

action $"Loading camera modules: " \
    modprobe pwc fps=15
# && \
# modprobe ov511
# && \
#action $"Configuring dynamic libraries: " /usr/sbin/ldconfig
#echo "alias pico='echo Please try jpico instead. Other editors include jmacs, joe, and vi.'" >>
/etc/bashrc

# Start thttpd web server
httpd start

# Mount the server for "long-term" image storage for later analysis
#action $"Loading NFS modules: " \
#insmod /usr/lib/modules/sunrpc.o && \
#insmod /usr/lib/modules/lockd.o && \
#insmod /usr/lib/modules/nfs.o
#action $"Mounting Server via NFS" \
#mount -t nfs server.crim:/home /mnt/nfs 2>&1 > /dev/console

```

Section 8.3.2.6 Linux TB-Interface Utilities C Source Code

Section 8.3.2.6.1 Basicx_interface

```
/* basicx_interface.c John M. Galeotti 7-9-01

*
* Interfaces to the BasicX on the NCSU CRIM's PC104 treaded base over a serial
* port softlinked to /dev/basicx. It opens the serial port and relays data
* between it and an input FIFO (/dev/basicx_in) and an output FIFO
* (/dev/basicx_out). This is necessary because the alternative (opening and
* closing the serial port for each utility call, many times a second), either
* drops or adds characters, but in any case always fails very quickly.
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

#define BUFF_SIZE 128

int main(void){
    int result, basicx, basicx_in, basicx_out;
    char buff[BUFF_SIZE];

    //fprintf(stderr, "Opening basicx\n");
    basicx = open("/dev/basicx", O_RDWR);
    if( -1 == basicx ){
        printf("\nError opening /dev/basicx, which should point to the treaded\n"
            "base's BasicX's serial port.\n\n");
        return 1;
    }
    //fprintf(stderr, "Opening basicx_in\n");
    basicx_in = open("/dev/basicx_in", O_RDONLY );
    if( -1 == basicx_in ){
        printf("\nError opening /dev/basicx_in, which should be a fifo. Make it with:\n"
            "mkfifo -m 660 /dev/basicx_in ; chgrp robot /dev/basicx_in\n\n"
            "/dev/basicx_out should be produced in a similar maner\n\n");
        return 1;
    }
    //fprintf(stderr, "Opening basicx_out\n");
    errno=0;
    basicx_out = open("/dev/basicx_out", O_WRONLY);
    if( -1 == basicx_out ){
        perror("basicx_out");
        printf("\nError opening /dev/basicx_out, which should be a fifo. Make it with:\n"
            "mkfifo -m 660 /dev/basicx_out ; chgrp robot /dev/basicx_out\n\n");
        return 1;
    }
    //fprintf(stderr, "Files Open\n");
    /* Now that we have the BasicX's serial port, configure it to our liking */
    system("stty sane 19200 -echo <> /dev/basicx");

    /*** Loop ***/

    while(1){

        //fprintf(stderr, "Loop Top\n");
        /* Send the next command to the BasicX */
        while(!( result = read(basicx_in, buff, BUFF_SIZE) )){
            usleep(10000); /* pause 1/100th of a second */
        }
        //fprintf(stderr, "Got In Data (%d)\n", result);
        result = write(basicx, buff, result);
        //fprintf(stderr, "Sent In Data (%d)\n", result);
        /* Read and tell the result */
        result = read(basicx, buff, BUFF_SIZE);
        //fprintf(stderr, "Got Out Data (%d)\n", result);
        result = write(basicx_out, buff, result);
        //fprintf(stderr, "Sent Out Data (%d)\n", result);
    }
}
```

```

    }

    /*** Close up shop ***/

    close(basicx);
    close(basicx_in);
    close(basicx_out);
    return 0;
}

```

Section 8.3.2.6.2 Readpin

```

/* readpin.c John M. Galeotti 7-5-01

 * "system utility" to read the value of a digital input pin on the NCSU CRIM's
 * PC104 treaded base.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define BUFF_SIZE 128

int main( int argc, char *argv[] ){
    char pinnum, pinchar;
    char result[BUFF_SIZE];
    FILE *basicx_in, *basicx_out;

    if(argc < 2){
        printf("\nUsage: readpin [pin number]\n\n"
            "Where [pin number] is the number (9-20) of the pin to read\n"
            "As long as [pin number] is >= 9, the pin will be set\n"
            "to high-impedance input before reading the digital\n"
            "value. If [pin number] is < 9, then the pin will be\n"
            "left as an output and its (almost meaningless) digital\n"
            "value will be read and returned.\n\n");
        return 1;
    }

    pinnum = (char)strtol( argv[1], (char**)NULL, 0 );
    if( (pinnum < 0) || (pinnum > 35) ){
        printf("Error: [pin number] must be between 0 and 35.\n");
        return 1;
    }
    pinchar = (pinnum > 9) ? (pinnum - 10 + 'A') : (pinnum + '0');

    /* We must open the input FIFO first or risk deadlocking the server */
    basicx_in = fopen("/dev/basicx_in","w");
    if( !basicx_in ){
        printf("Error opening /dev/basicx_in. Is basicx_interface running?\n\n");
        return 1;
    }
    basicx_out = fopen("/dev/basicx_out","r");
    if( !basicx_out ){
        printf("Error opening /dev/basicx_out. Is basicx_interface running?\n\n");
        return 1;
    }

    /* We MUST use unbuffered FIFO output */
    setvbuf(basicx_in, NULL, _IONBF, 0);

    fprintf(basicx_in, "R%c", pinchar);

    fgets(result,BUFF_SIZE,basicx_out);
    if( 'R' == result[0] ){
        printf("%c\n",result[1]);
    }else{
        printf("Warning: Intercepted somebody else's response:\n %s",result);
    }

    fclose(basicx_in);
    fclose(basicx_out);
}

```



```

    return 0;
}

```

Section 8.3.2.6.3 Setpwm

```

/* setpwm.c John M. Galeotti 7-2-01

 * "system utility" to set the PWM speed of both treads on the NCSU CRIM's
 * PC104 treaded base. Talks to the basicx_interface server.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define BUFF_SIZE 128

int main( int argc, char *argv[] ){
    unsigned char direction, leftdir, leftval, rightdir, rightval;
    FILE *basicx_in, *basicx_out;
    char result[BUFF_SIZE];

    if(argc < 5){
        printf("\nUsage: setpwm leftdir leftval rightdir rightval\n\n"
            "Where leftdir and rightdir are one of f,F,b,B and\n"
            "      leftval and rightval are [0-255] and can be given\n"
            "      in hex by prefixing 0x or in octal by prefixing 0\n\n");
        return 1;
    }

    leftdir=argv[1][0];
    if( 'f' == leftdir || 'F' == leftdir){
        leftdir=1;
    }else if( 'b' == leftdir || 'B' == leftdir){
        leftdir=0;
    }else{
        printf("Error, Invalid argument %c for leftdir\n",leftdir);
        return 1;
    }

    rightdir=argv[3][0];
    if( 'f' == rightdir || 'F' == rightdir){
        rightdir=1;
    }else if( 'b' == rightdir || 'B' == rightdir){
        rightdir=0;
    }else{
        printf("Error, Invalid argument %c for rightdir\n",rightdir);
        return 1;
    }

    direction = (leftdir << 1) | rightdir;

    leftval = (char)strtol( argv[2], (char**)NULL, 0 );
    rightval = (char)strtol( argv[4], (char**)NULL, 0 );

    /* We must open the input FIFO first or risk deadlocking the server */
    basicx_in = fopen("/dev/basicx_in","w");
    if( !basicx_in ){
        printf("Error opening /dev/basicx_in. Is basicx_interface running?\n\n");
        return 1;
    }
    basicx_out = fopen("/dev/basicx_out","r");
    if( !basicx_out ){
        printf("Error opening /dev/basicx_out. Is basicx_interface running?\n\n");
        return 1;
    }

    /* We MUST use unbuffered FIFO output */
    setvbuf(basicx_in, NULL, _IONBF, 0);

    fprintf(basicx_in, "%c%c%c", direction, leftval, rightval);

    fgets(result,BUFF_SIZE,basicx_out);

```

```

if( 'P' != result[0] ){
    printf("Warning: Intercepted somebody else's response:\n %s",result);
}

fclose(basicx_in);
fclose(basicx_out);

return 0;
}

```

Section 8.3.2.7 Other Source Code and Binary Files

The remainder of Infinite Atom consists of standard files for a Linux distribution, with the exception that many configuration files have been modified to work on the EvBot and most binary files have had their symbols stripped to reduce their size. Section 4.4.3.3 explains how the contents of Infinite Atom were chosen. The files were mostly taken from standard Red Hat distributions, but some were either obtained elsewhere or written from scratch. Such “nonstandard” files are included in either source or binary form on the CD-ROM in the “EvBot Software\Infinite Atom\Special nonstandard components” directory. The README.txt file on the CD-ROM contains important details and restrictions concerning some of the software on the CD-ROM.

Section 8.3.3 EvBot MATLAB Controller Source Code

Complete source code for the EvBot controllers used in the experiments by Ph.D. student Andrew Nelson is included on the CD-ROM in the “EvBot Software\MATLAB controllers” directory. The code for the tactile controller is in the “matlab_tactile_controller” subdirectory and the code for the vision-based controllers is in the “matlab_vision_controllers” subdirectory. The vision-based controllers share a common code base and the particular controller used is chosen by setting the value of the robot.controller.type variable in the file drive_robot.m. All the code is self-documented with comments.

Section 8.4 EvBot Support Environment

Section 8.4.1 EvBot Development Station

In order to facilitate software installation on a new EvBot, as well as future EvBot-specific software development and modification, it is necessary to have an EvBot development station. The development station consists primarily of the following components attached to an EvBot:

- Two IDE hard drives with cable
- 1.44 MB Floppy drive with cable
- PC/104 video card and monitor
- Keyboard
- Computer power supply
- USB mouse optional

Combined, these components can turn an EvBot into a desktop computer. Attachment of most of the devices is rather straight-forward. Simply follow the MZ104's instructions, using the connectors on the UB where appropriate. The computer power supply is for powering the disk drives. The video card should be attached to the top of the PC/104 stack, and the USB camera must not be installed at the same time as the video card because it physically blocks the video card and draws too much power from the EvBot's power supply if a typical video card is also attached. The first hard drive needs to be prepared, but the second hard drive can be left blank if desired. The second drive must be attached to the EvBot to make the MZ104 boot properly from the first hard drive when the MZ104 is setup to boot from the DOC in the absence of a hard drive (as it

should be). The first hard drive should be prepared by installing Linux (preferably Red Hat 7.2) on it and then adding some software for use with the EvBot.

It is probably easiest to install Linux on the first hard drive by attaching both hard drives to an EvBot and then doing a network-based installation onto the first hard drive. It is also possible, however, to attach the first hard drive to another computer to install Linux on it before attaching it to an EvBot, but because complications can easily arise this method will not be further discussed. The Linux installation should be a full install, if possible. After installation completes, additional software needs to be installed on the hard drive for use with the EvBot.

Two directories need to be created on the hard drive to hold original copies of the contents of an EvBot's filesystems for installation onto EvBots as they are built. It is highly recommended that these directories be named /doc_image and /ATA_image. The contents of the EvBot's filesystems should then be placed in these directories by extracting them from the archives included on the CD-ROM in the "EvBot Software\Infinite Atom\Filesystem contents" directory. It will probably be easiest to make the archives network accessible and then use "tar xzvf <complete path and name of archive> -C <destination directory>" to extract DOC_filesystem_contents.tgz to /doc_image (use /doc_image as <destination>) and ATA_flash_PC-Card_filesystem_contents.tgz to /ATA_image (use /ATA_image as <destination>).

Because MATLAB is commercial software, it was not included on the CD-ROM and must be added to the /ATA_image directory according to the procedure below. Begin by installing MATLAB onto a normal Linux desktop computer. The root account

should be used for the installation. Do a minimal install with the target of either / or /usr. After installation completes, remove the files and directories listed in Section 8.3.2.1.8 and then copy everything from MATLAB's root directory to /ATA_image/matlab on the development station's hard drive, being sure not to erase the files already there. Be aware that every installed copy of MATLAB must be legally licensed, including those copies placed onto EvBots by copying from the development station's hard drive.

A Linux kernel with DOC support also needs to be added to the development station's hard drive. This can be done by extracting the kernel_2.4.14.tgz archive located on the CD-ROM in the "Support Software\EvBot development station" directory to / on the hard drive (use "tar xzvf <full path to archive>/ kernel_2.4.14.tgz -C /) and then running the lilo command on the hard drive.

Finally, a robot account needs to be created on the development station's hard drive. The account should have both the user ID and the group ID of 501. Consult the manual for the distribution of Linux installed on the hard drive for instructions on how to do this. It is highly recommended that the robot account be either the first or second account created on the hard drive after the root account has been created. Once the robot account has been added, the etc/passwd file in /doc_image needs to be replaced by the /etc/passwd file on the hard drive to change the passwords used by future EvBots to those set for the development station. This can be done by typing the command "/bin/cp -a /etc/passwd /doc_image/etc/" on the development station.

In addition to all the components mentioned for the development station, a serial console of some kind will also be necessary. This will typically be a computer running a

terminal emulation program such as HyperTerminal for Windows or minicom for Linux. The computer must use a null-modem cable to connect its serial port to the second serial port of the EvBot. Please note that both of the EvBot's serial ports are five-by-two pin headers, and an adapter cable is necessary to attach a conventional serial cable to an EvBot. Such an adapter cable can either be purchased (from Tri-M or other computer hardware vendors) or built using the MZ104's manual as a guide. If the adapter cable is built, it may be beneficial to build the entire null-modem cable and make the adapter an integral part of the cable (i.e. make one end of the null-modem cable attach directly to a five-by-two pin header).

Although not necessary, a keyboard, video, mouse (KVM) switch may also be very useful, allowing a single keyboard, monitor, and mouse to be used both for the EvBot development station and for the serial console. Also, it may be helpful to attach a cable with a 2.1 mm DC power plug to a 7.2 V, 2 A power supply for powering the EvBot for extended periods of time when untethered mobility is not required.

Section 8.4.2 Server Setup

EvBot colony operation and setup can be simplified by using a server to automatically configure the EvBots' networks and supply the IP address for the EvBots' DNS names. The server must be in contact with the EvBots' wireless network, either via its own wireless Ethernet connection or via a wireless access point attached to the server's standard wired network. The copy of Infinite Atom included on the CD-ROM has been configured to rely on the presence of such a server on the EvBot's wireless network, although it could be modified by a knowledgeable user to have such networking

information hard-coded into each EvBot. To reduce cost and simplify operation, the server can be the same computer used as the serial console for the EvBot development station. In such a case, it should have a modern version of Linux installed on it. It should have a DNS server setup on it to match the EvBots' names with their IP address. Once the DNS server is working, the server should also have a DHCP server setup on it to provide each EvBot with its IP address and other networking settings every time each EvBot boots up. The procedure for setting up a DNS server and a DHCP server should be specified in the user's manual or the administrator's manual accompanying the Linux distribution installed on the server. It may also be useful to setup remote data logging facilities for the EvBots on the server, possibly by using the HTTP protocol's put command. Consult the Apache web server's manual for information on setting this up.

Section 8.4.3 Maze Environment Detailed Description

To assist in the creation of useful experiments, a semi-modular closed-maze environment was created for the EvBots (see Figure 8.30). The maze consisted of black wall sections and vertices resting on an aqua floor. Each wall section measured 29.5 inches long by 1.5 inches wide, and each vertex measured 1.5 inches square, making each cell of the maze a square measuring 31 inches on the side. By adding and removing wall sections and vertices, the maze could be configured to any desired size and pattern. For the experiments conducted, the maze measured five cells on each side. The walls were nine inches high, which was one inch taller than the height of an EvBot. To maintain a consistent environment for consecutive vision-based experiments, a visual barrier made of foam-core was placed around the outside walls of the maze. The barrier helped

regulate maze lighting and prevented EvBots from reacting to people in the laboratory. It was reversible, painted medium blue on one side and sky blue on the other. The high contrast between the maze walls and their surroundings allowed each EvBot to accurately determine its distance to a wall simply by counting how many pixels high the wall appeared above the horizon in the EvBot's field of view. Given the wall's actual fixed height above the horizon, its apparent height is all that is needed to compute the distance to the wall by using simple trigonometry. This method of distance calculation required only a single camera and was most accurate at short range.

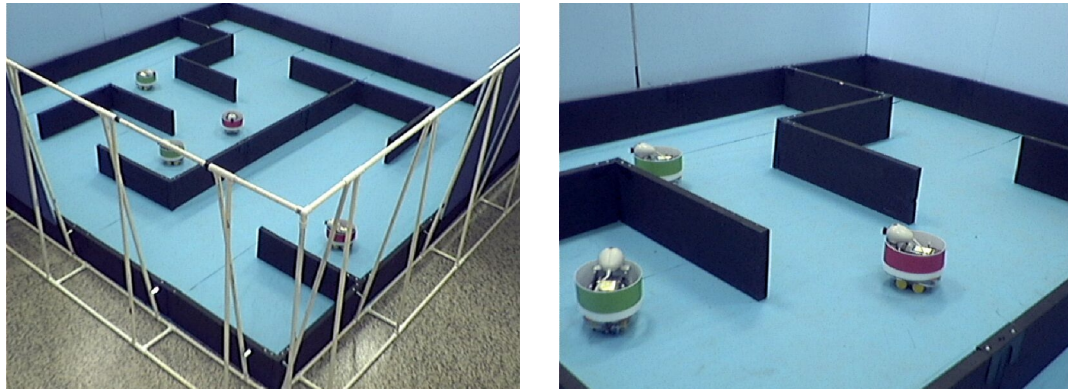


Figure 8.30 The maze environment

An overhead camera with a fisheye lens was mounted in an approximately centered position above the maze so that it was able to see the entire maze environment. It was connected to a computer's video capture card, allowing accurate video and images of the maze environment to be captured. For most of the experiments, web cam software was used to periodically capture images from the maze at a rate faster than the control loop of the EvBots, allowing every position at which every EvBot stopped to be recorded. Figure 8.31 shows a sample image acquired from the camera during such an experiment.

Notice the fisheye distortion, which could theoretically be corrected by a computer (although it was not). The black corners are the cylindrical wall of the fisheye lens.

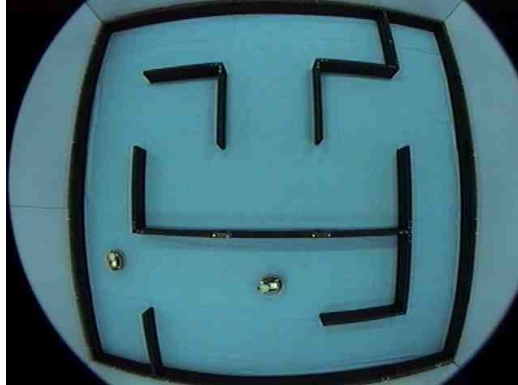


Figure 8.31 Image of the maze acquired by its overhead camera during an experiment

Section 8.4.4 Simulation Environment

Andrew Nelson's MATLAB simulation environment used to genetically train EvBot controllers for the maze navigation experiment involving binary tactile sensors is included on the CD-ROM in the "Support Software\EvBot simulator\simulator to train artificial neural networks" directory. To train a new population of controllers, use the `simulate_world.m` MATLAB script. Periodically, the script will save the current population of controllers (with the best controller first) to the `last_evolved_robots.mat` file. Once the controllers perform adequately well in simulation, they can be transferred to an EvBot by copying the file to the EvBot and modifying the `drive_robot.m` file in the EvBot's existing tactile controller directory (`/matlab/ann_first_working`) to point to the new file of controllers. Since the EvBot does not run the tactile controller by default, MATLAB will have to be stopped and restarted to use the tactile controller. Optionally, the EvBot can be made to run the tactile controller by default by changing

/matlab/startup.m to call /matlab/ann_first_working/drive_robot.m (more details on changing the default controller are provided in Section 8.6). In either case, once drive_robot.m is called, it will automatically use the best controller stored in the .mat file provided.

The simulator itself is a discretized time-step, two dimensional cellular environment with real valued robot positions. It models robot kinetics with two-wheeled differential steering and simulates binary tactile sensors. Users wishing to see a demo of a single EvBot navigating through a maze in simulation can also run the demo.m script in the “Support Software\EvBot simulator\demo of simulated robot in maze” directory on the CD-ROM. A sample output image from the simulation is provided in Figure 8.32.

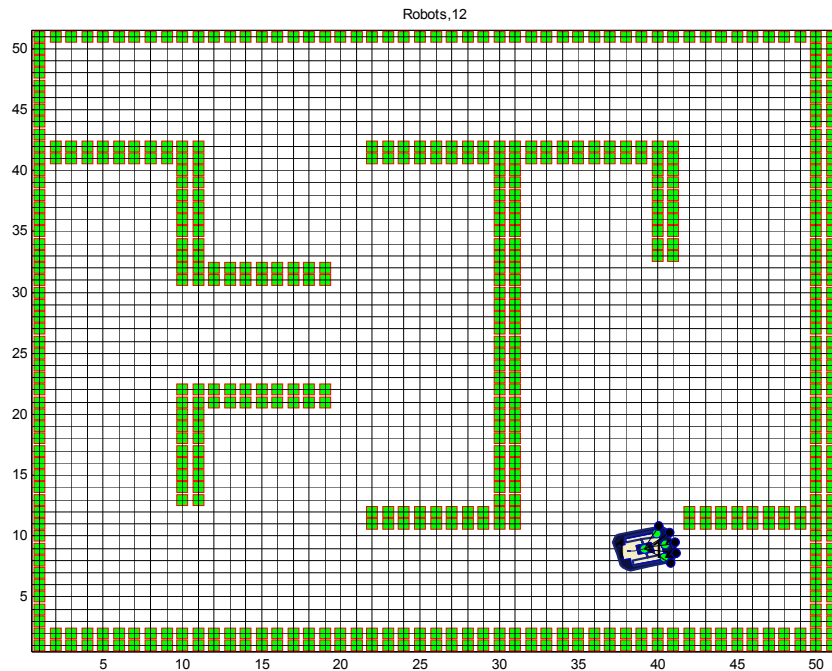


Figure 8.32 The MATLAB simulation demo showing a single EvBot navigating through a maze

Section 8.5 Setup and Configuration Procedure for a New EvBot

Make sure there are no AA batteries in the TB. The EvBot needs to be attached to the development station to download the software. For the initial boot, attach only the power supply, floppy drive, keyboard, and PC/104 video card (with monitor attached) to the EvBot. Be sure that the red wire on the disk cable goes to pin one on the motherboard connector. Also, make sure that the video card is fully seated against the standoffs; there should be no space between the top of the standoffs and the bottom of the video card. Connect the serial console to the robot's COM2 serial port. Power up the drives' power supply before turning on the EvBot. Immediately after turning on the EvBot, watch for the message "Press <F2> to enter setup" to appear at the bottom left corner of the screen. When the message appears, press the F2 key on the keyboard. When the PhoenixBIOS Setup Utility loads, set the time and date; this is *critical* for running MATLAB. Under the "Advanced" menu, enable "Reset Configuration Data" by setting it to yes and then in the "Console Redirection" submenu, set the COM Port Address to "On-board COM B," "Console Type" to "VT100," and "Continue C.R. after POST" to on. Then, press the Esc key to return to the "Advanced" menu. Enter the "I/O Device Configuration" submenu and set both "Serial port A" and "Serial port B" to "Enabled." Insert the DOC 5.0 Update Disk (see Section 8.5.1) into the floppy drive and then press the F10 key to save the settings and reboot the machine. Press <Enter> when asked to confirm saving and rebooting. When the disk finishes booting, type the command "dformat /win:d000 /s:doc50.exb /noformat /ebda" at the prompt and press <Enter>.

When `dformat` finishes and requests that the machine be rebooted, turn off the EvBot, turn off the drives' power supply, and then attach the hard drive cable *in that order*. Disconnect the floppy cable, and then turn on the power supply and the EvBot. Linux will boot normally by default; if hardware detection (`kudzu`) asks about changing a hardware configuration, select "keep configuration." Login as root and type the following commands to install Infinite Atom on the EvBot's DOC and ATA flash card:

- `modprobe doc`
- `mke2fs -m0 /dev/msys/fla1`
- `mke2fs -m0 /dev/hde1`
- `mount /mnt/doc`
- `mount /mnt/flash`
- `cp -av /doc_image/* /mnt/doc/`
- `cp -av /ATA_image/* /mnt/flash/`
- `doc-lilo -r /mnt/doc`
- `halt`

As soon as the message "System Halted" appears on either the monitor or the serial console, turn off the EvBot. Turn off the drives' power supply and disconnect the hard drive. Turn on the EvBot to confirm its operation. After the EvBot finishes booting ("Starting crond: [OK]" will be the last line to appear on the screen), no obvious errors should have been reported, and three lines similar to the following should be visible at different places on the screen:

- `cardmgr[108]: executing: './ide start hde'`

- cardmgr[108]: executing: './network start eth0'
- Starting sshd: [OK]

Turn off the EvBot. Before attaching the camera, disconnect the video card because the EvBot's power supply does not have enough capacity to power both the video card and the USB camera at the same time. It is also recommended that the camera be focused while attached to another computer before attaching it to the EvBot. After disconnecting the video card and unplugging the keyboard, attach the USB camera to the camera-mounting plate as shown in Figure 8.21 and Figure 8.27 by screwing an appropriate screw through the plate into the camera's tripod mount. As shown in Figure 8.27, wrap the USB cord around the standoffs between the TB and the UB, being careful not to pull the cord too tightly against the wireless Ethernet card. After wrapping the cord all the way up, loop the end of the cable around the beginning of the cable as shown in Figure 8.27 before plugging the cable into the UB's USB connector.

Section 8.5.1 Creation of the DOC 5.0 Update Disk

The DOC 5.0 Update Disk can be created by copying some files to a Windows or DOS boot disk. Using either operating system, prepare a blank 1.44 MB floppy disk by typing the command "format a: /s" at the command prompt and then pressing <Enter>. After formatting completes, copy all of the files in the "Support Software\EvBot development station\DOC 5.0 Update Disk" directory on the CD-ROM to the floppy disk. When copying completes, the disk is ready. Due to licensing restrictions imposed by M-Systems, all DOC software, including the software on the DOC 5.0 Update Disk, can only be used with DOC products. Refer to the README.txt file on the CD-ROM.

Section 8.5.2 ESCD Reset Procedure

Two different versions of the MZ104 were used to build the experimental EvBot colony. Two of the first version were purchased to develop the EvBot platform and the others were purchased at a later date when the newer, second version was available. Because the first version lacked a BIOS boot setting to reset the MZ104's Extend System Configuration Data (ESCD), it was possible to misconfigure the ESCD by disconnecting the CMOS battery after both the CMOS and ESCD values had been set. This is because removing the battery will clear all user-settable CMOS settings (which devices are enabled and how they are configured) but will not clear the ESCD settings (which resources are assigned to each device); resources would then be assigned to nonexistent devices, preventing attached devices from using them. Since ESCD settings could not be directly set in the CMOS setup menus, a difficult situation was created. This was not a problem for the newer versions of the MZ104, since they had the CMOS option to have all ESCD values reset on bootup, but a solution was necessary for the older versions. Through extensive experimentation, the following sequence of actions was found that would almost always restore the ESCD to working values (The EvBot should be properly *shutdown* after *each* of the following steps):

1. Boot the EvBot without the PC/104 PC-Card interface installed.
2. Boot the EvBot with the hard drive and the PC/104 PC-Card interface installed. The ATA flash card should be installed above the wireless Ethernet card. It is expected that one or both of the PC-Cards will not work at this point.

3. Boot the EvBot with the hard drive attached but without any cards in the PC-Card interface. After Linux finishes booting, insert the wireless Ethernet card into the bottom slot of the PC-Card interface, and wait for Linux to recognize it.
4. Remove the 3 V battery from the UB. After a few minutes reinsert it and then boot the EvBot with the hard drive and PC-Cards attached as in the second step above. Both PC-Cards should work now.
5. Boot the EvBot with the hard drive attached, but as soon as the message “Press <F2> to enter setup” appears at the bottom left corner of the screen, press the F2 key to enter the PhoenixBIOS Setup Utility. Enter the “Advanced” menu and then the “I/O Device Configuration” submenu; change the first serial port to the “auto” setting. Press the F10 key to save the settings and reboot the machine (press <Enter> when asked to confirm saving and rebooting). Allow Linux to finish loading before shutting it down and proceeding with the next step.
6. Boot the EvBot *without* the hard drive attached. Everything should work properly now.

Section 8.6 Procedure for Adding a New Controller to an EvBot

It is fairly simple to add a new controller to an EvBot. If the controller is a MATLAB controller, then it should be placed in its own subdirectory within the /matlab directory. Also, a startup.m file should be placed in the /matlab directory that will

automatically change to the controller's directory and then run the controller. Actually, it is recommend, but unnecessary, that rather than placing the startup.m file in /matlab, it should be placed in the controller's subdirectory. The controller's subdirectory could then be symbolically linked to /matlab/robot_controller and /matlab/startup.m could be a symbolic link to robot_controller/startup.m. The advantage of following this more complex procedure is that complex startup.m files can be preserved for each controller, and any of the controllers stored on an EvBot can be made the default controller simply by making the /matlab/robot_controller symbolic link point to the desired controller's subdirectory. Because /matlab/startup.m symbolically links through the robot_controller symbolic link, it never needs to be updated. For the reasons given, this more complex symbolically linked layout is used on the version of Infinite Atom included on the CD-ROM. However, the simple procedure of just editing /matlab/startup.m can still be used to make any new controller work if the benefits of the more complex procedure are not desired. In any case, the new controller will be used automatically the next time the EvBot boots, or it can be used immediately by restarting MATLAB by using the stopautodrive command followed by the startautodrive command.

It is slightly more difficult to add a new controller to an EvBot if that controller is *not* MATLAB based. A separate directory should be made in /usr to hold the controller. Then, while logged into the EvBot as root, the /usr/etc/robot_controller symbolic link should be changed to point to the new controller. The command “ln -sf <full path to the new controller> /usr/etc/robot_controller” will accomplish this. Should the new controller ever exit or die, it will be automatically restarted by Infinite Atom. (The

stopautodrive command should be used to stop the controller when such is desired so that Infinite Atom will not automatically restart it.)

Whether or not a new controller is MATLAB based, if it requires that new drivers be loaded, the appropriate command(s) can be placed in the `/usr/etc/rc.local` file. Of course, if the drivers are not already on the EvBot (which is probably the case), then the drivers must also be copied onto the EvBot. It is also possible to start system services and other background commands on bootup by adding commands to start them to `/usr/etc/rc.local`. In fact, any command placed in the file will be automatically run on bootup. For this reason, the file can *only be edited by the root login ID*.